SOFTWARE ENGINEERING LABORATORY SERIES SEL-82-004

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME I

JULY 1982

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-82-004

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME I

JULY 1982

NASA
National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. The papers contained in this document appeared previously as indicated in each section.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582.1
NASA/GSFC
Greenbelt, Maryland 20771

TABLE OF CONTENTS

SECTION 1 – INTRODUCTION

SECTION 1 - INTRODUCTION

This document is a collection of technical papers produced by participants in the Software Engineering Laboratory (SEL) during the 5-year period ending December 31, 1981. The goal of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. Although these papers cover a wide range of topics related to software engineering, they do not completely describe the activites and interests of the SEL. Additional information about the SEL and its research efforts can be obtained from the sources listed in the bibliography.

For the convenience of this presentation, the 10 papers are organized into 4 major topics, as follows:

- The SEL organization (Section 2)
- Resource models (Section 3)
- Software measures (Section 4)
- Software engineering applications (Section 5)

Although these topics are interrelated, some general distinctions among them can be made. The first topic discussed is the organization, objectives, and operation of the SEL itself. Then some research results in the areas of defining and evaluating resource models and software measures are presented. The last section includes discussions of the application of resource models and software measures to software management and the evaluation of software technology.

The SEL is still actively working to understand and improve the software development process at Goddard Space Flight Center. Future efforts will be documented in additional volumes of the Collected Software Engineering Papers and in other SEL publications.

SECTION 2 – THE SEL ORGANIZATION

SECTION 2 - THE SEL ORGANIZATION

The technical papers included in this section were originally published as indicated below:

- Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977 (reprinted by permission of the authors)
- Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Life Cycle Management Workshop, September 1977 (reprinted by permission of the authors)

The Software Engineering Laboratory: Objectives

Victor R. Basili
Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Md. 20742

I. INTRODUCTION

A great deal of time and money has been and will continue to be spent in developing software. Much effort has gone into the generation of various software development methodologies that are meant to improve both the process and the product ([MYER, 75], [BAKE, 74], [WOLV, 72]). Unfortunately, it has not always been clear what the underlying principles involved in the software development process are and what effect the methodologies have; it is not always clear what constitutes a better product. Thus progress in finding techniques that produce better, cheaper software depends on developing new deeper understandings of good software and the software development process through studying the underlying principles involved in software and the development process. At the same time we must continue to produce software.

To gain a better knowledge of what is good in the current methodologies and what is still needed, and to help understand the underlying principles of the software development process, we must analyze current techniques, understand what we are doing right, understand what we are doing wrong, and understand what we can change.

There are several ways of doing this. One way is to analyze the development process and the product at various stages of development. Unfortunately, such analysis is a tedious process. But it must be performed if we are to gain any real insight into the problems of software development and make improvements in the process. We need to study carefully the effect of various changes in the development process or the product to determine whether or not a particular methodology has any real effect, and more importantly, what kind of effect ([THAY, 76], [WALS, 77]).

This requires measures of all kinds, quantifiable and nonquantifiable. Nonquantifiable measures, although subjective, reveal a great deal of information about the product. We can "see" good design and code that meets the problem requirements in a clear, understandable,

effective way and is easy to modify and maintain in unforeseen circumstances. This kind of understanding is clearly needed, and clearly fruitful; it is accomplished by reading and understanding the design and code. Unfortunately, these judgements are not easy to quantify. They require a great deal of time to analyze and measure each product, or class of products.

A secondary approach is to develop a set of measures that attempt to quantify these qualitative characteristics of good software design and development. Although there is currently no mechanical way of evaluating design, the development of quantitative measures that correlate well with subjective judgements of quality can aid in the understanding and evaluation of the product and process. For example, the "goodness" of a product is related to the time it takes to modify it and the aspects of its organizational structure that permit ease of modification and ease of finding and correcting errors where ease is measured in terms of the time required, number of places code needs to be changed, etc. The "goodness" of the development methodology is related to the "goodness" of the product it produces, e.g., the number and difficulty of finding errors in the product it produces.

It is important to understand what characterizes classes of problems and products, what kinds of problems are encountered and errors made in the development of a particular class of products, whether or not a particular methodology helps in exposing or minimizing the number or effect of a class of errors, what the relationship is between methodology and management control, estimating, etc. A better understanding of the factors that affect the development of software and their interrelationships is required in order to gain better insights into the underlying principles. The Software Engineering Laboratory has been established, in August 1976, at NASA Goddard Space Flight Center in cooperation with the University of Maryland to promote such understanding. The goals of the laboratory are to analyze the software development process and the software produced in order to understand the development process, the software product itself, the effect of various "improvements" on the process with respect to the methodology, and to develop quantitative measures that correlate well with intuitive notions of good software.

The next section gives an overview of the research objectives and experiments being performed at the Laboratory. Section III contains the current list of factors that affect the software development process or product and are to be studied or neutralized. The data collection and data management activities are discussed

in Section IV. The last section contains information on the current status and future plans for the Laboratory. Further details of this project can be found in [BASI, 77].

II. ACTIVITIES

It is clear that many kinds of data can be gathered and analyzed to develop quantitative information about the software process and the product to which it leads. The laboratory has limited funding and personnel and for this reason has limited its scope to studying three very specific areas related to reliability, management, and complexity. It is expected that the scope will eventually expand as we learn more about the collection of valid data and what can be done with it. In this section we discuss the research activities and the two classes of experiments to be run.

Because error-free software is as yet an unattainable goal, the reliability study will provide insight into the nature and causes of software errors. We would like to classify errors, expose techniques that reduce the total number or classes of errors, and detect the effect or lifetime of these errors ([SHOO, 75], [THAY, 76], [ENDR,75], [GANN, 75], [AMOR, 73]). We expect to detect the point at which errors enter the process and the relative costs of finding and fixing them.

Management of the software development process is as poorly understood as the technology involved. We believe that a major effort should be expended on this area. The management aspect of the Software Engineering Laboratory involves the analysis of the management process, the classification of projects from a management point of view and the development of reasonable management measures for estimating time, cost, and productivity ([BAUM, 63], [TAUS, 76]). We will study the effect of various factors, such as time, money, size, computer access, techniques, tools, organization, standards, milestones, etc. We would like to understand at what point in the development process, estimates become reasonably accurate, how one can measure good visibility and management control and under what conditions certain methodologies help provide management control.

Lastly, there is a relation between the development methodology and the product it produces. A good methodology should help produce a less complex product than a "bad" one. We are trying to discover whether the complexity of a software system can be measured by the structure of the resulting programs ([SULL, 73], [HELL, 72], [VANE, 70]). Do various techniques create a more systematic structure, one that is easier to read and

maintain, where data and function are localized with a minimal amount of interaction between modules? The relationship between various complexity measures of program structure will be examined throughout the development process and such measures as error rate, development time, the accuracy and speed of modification will be correlated with these complexity measures.

Two kinds of experiments are being conducted: screening experiments and controlled experiments. In the screening experiments, we are collecting data on a large assortment of projects of varying sizes and types. The impact on the development process is manifested by the requirement that the developers fill out a set of data collection forms (see Section IV). The purpose of the screening experiments is to determine how software is developed now. We are organizing a data bank of information to classify projects for future reference and public availability, analyze what methodologies are being used as opposed to what methodologies are supposed to be used, demonstrate how carefully the actual implementation of a methodology can be monitored, discover what parameters can be validly isolated, expose the parameters that appear to be causing major problems, and discover the appropriate milestones and techniques that show success under certain conditions. While the data collected in the screening experiments may not be complete or totally accurate, it will provide input for the more strictly monitored controlled experiments.

The purpose of the controlled experiments is to discover the effect of various factors on the software development process and product in a reasonably controlled environment. A set of duplicate developments will be performed and detailed data collected for all of them. A carefully chosen set of techniques will be taught to and used by one of the development groups, denoted as the "impacted" group. We will then analyze the effect of the introduced factors by comparing the impacted development process and product in a reasonably controlled environment.

The experiment must be designed in such a way as to insure that we are testing the real hypothesis, i.e., to guarantee that we are measuring what we think we are measuring. It is important that all the contributing factors be well understood and the factors that we are not studying be neutralized [CAMP, 66]. Our approach is first to develop a particular experimental design, analyze its ability to neutralize potential interfering factors, (i.e., individual programmer capability) and perform one experiment. Based on this experience, the design will be modified and experiments repeated until we have arrived at a reasonable standard.

One current experimental design is to have two groups, Group 0 and Group 1, each develop a product, A. We will then impact Group 1 with a set of factors by teaching them the use of certain development techniques. Both groups will then develop a second small project B to give Group 1 some experience with the techniques in an operating environment. Then both groups will develop product C, Group 1 using the new approach. This gives us several points of comparison. We can discover any difference in personnel by comparing project A for both groups; the two groups can then be more honestly compared in project C by factoring out differences from project A. The measures developed for the areas of interest will be used to compare the two processes and products.

In a second controlled experiment, several large scale projects (5 to 10 man years each) are to be carefully monitored with some of the personnel given a training course and set methodology to use. Using the notation above, these will be a set of C projects with no A and B. While the projects are not identical, they are highly similar and should yield information about differences in techniques. In Section V, both of these controlled experiments will be described in greater detail.

III. FACTORS

There are a large number of factors that affect the software development process and software product. Initially, we are interested in a list of potential factors to establish the kind of data that needs to be collected. Next, we are interested in the kinds of factors that we can reliably measure. From this measurable set of factors, we would like to isolate those that appear to have a major impact on the development process and product, i.e., those whose use or non-use show large variation in our measures. Finally, when we have a better understanding of the factors affecting the software development process, we want to quantify them in some way by perturbing them to study their effects or neutralizing them to make sure they are not affecting factors that are under study.

Our procedure is to start with as complete a list of factors and categories of factors as possible. We expect continually to build, iterate, and refine this list through the activities of the laboratory. The development of reporting forms and automated tools have helped define the list of factors that we can isolate. The screening experiments will help further isolate those factors which we can measure and those that appear to be contributing strongly to the various measures associated with errors, complexity of program structure,

management difficulties, etc. The controlled experiments will be used to demonstrate the effect of the various factors that have been shown worth isolated study.

A list of factors is given below, categorized by their association to the problem, the people, the process, the product, the resources, and the tools. Some factors may fit in more than one category but are listed only once.

A. People Factors: These include all the individuals involved in the software development process including managers, analysts, designers, programmers, librarians, etc. People related factors that can affect the development process include: number of people, level of expertise of the individual members, organization of the group, previous experience with the problem, previous experience with the methodology, previous experience with working with other members of the group, ability to communicate, morale of the individuals, and capability of each individual.

B. Problem Factors: The problem is the application or task for which a software system is being developed. Problem related factors include: type of problem (mathematical, database manipulation, etc.), relative newness to state of the art requirements, magnitude of the problem, susceptibility to change, new start or modification of an existing system, final product required, e.g., object code, source, documentation, etc., state of the problem definition, e.g., rough requirements vs. formal specification, importance of the problem, and constraints placed on the solution.

C. Process Factors: The process consists of the particular methodologies, techniques, and standards used in each area of the software development. Process factors include: programming languages, process design languages ([VANL, 76]), specification languages, use of librarian ([BAKE, 75]), walk-throughs ([BAKE, 75]), test plan, code reading, top down design, top down development (stubs), iterative enhancement ([BASI, 76]), chief programmer team ([BAKE, 75]), Chapin charts, HIPO charts ([STAY, 76]), data flow diagrams, reporting mechanisms, structured programming ([MILL, 72], [DAHL, 72]), HOS techniques ([HAMI, 76]), and milestones.

D. Product Factors: The product of a software development effort is the software system itself. Product factors include: deliverables, size in lines of code,

words of memory, etc., efficiency tests, real-time requirements, correctness, portability, structure of control, in-line documentation, structure of data, number of modules, size of modules, connectivity of modules, target machine architecture, and overlay sizes.

E. Resource Factors: The resources are the nonhuman elements allocated and expanded to accomplish the software development. Resource factors include: target machine system, development machine system, development software, deadlines, budget, and response and turnaround times.

F. Tool Factors: The tools, although also a resource factor, are listed separately due to the important impact they have on development. Tools are the various supportive automated aids used during the various phases of the development process. Tool factors include ([REIF, 75], [BOEH, 75], [BROW, 73]): requirements analyzers (e.g., PSL/PSA [TEIC, 77], system design analyzers, source code analyzers (e.g., FACES [RAMA, 74]), database systems (e.g., DOMONIC [DOMO, 75]), PDL processors, automatic flowcharters, automated development libraries, implementation languages, analysis facilities, testing tools ([RAMA, 75], [MILL, 75]), and maintenance tools.

IV. Data Collection

Data collection occurs as four components - reporting forms, interviews, automatic collection of data by computer, and use of automated data analysis routines.

A. Forms: There are seven forms that were defined to obtain information on the factors given in Section III. These forms are filled out by various members of the project development team and are used to gather information at various states of the development process. They reveal the resource estimates at inception, the overall layout of the system, the updating of the estimates and the achievement of milestones, the time spent in various activities, the expenditures of resources, and an audit of all changes to the system. Several redundancy checks have been included to validate the accuracy of the information obtained.

Briefly, the seven forms are as follows (See Appendix 2 of [BASI, 77]):

1. The General Project Summary - This form is used to classify the project and will be used in conjunction with the other reporting forms to

measure the estimated versus actual development progress. It is filled out by the project manager at the beginning of the project, at each major milestone, and at the end. The final report should accurately describe the system development life cycle.

2. The Programmer/Analyst Survey - This form is to classify the background of the personnel on each project. It is filled out once at the start of the project by all personnel.

3. The Component Summary - This form is used to keep track of the components of a system. A component is a piece of the system identified by name or common function (e.g., an entry in a tree chart or baseline diagram for the system at any point in time, or a shared section of data such as a COMMON block). With the information on this form combined with the information on the Component Status Report, the structure and status of the system and its development can be monitored. This form is filled out for each component at the time that the component is defined, at the time it is completed, and at any point in time when a major modification is made. It is filled out by the person responsible for that component.

4. The Component Status Report - This form is used to keep track of the development of each component in the system. The form is turned in at the end of each week and for each component lists the number of hours spent on it. This form is filled out by persons working on the project.

5. The Resource Summary - This form keeps track of the project costs on a weekly basis. It is filled out by the project manager every week of the project duration. It should correlate closely with the component status report.

6. Change Report - The change report form is filled out every time the system changes because of change or error in design, code, specifications or requirements. The form identifies the error, its cause and other facets of the project that are affected.

7. Computer Program Run Analysis - This form is used to monitor the computer activities used in the project. An entry is made every time the computer is used by the person initiating the run.

B. Interviews: Interviews are used to validate the accuracy of the forms and to supplement the information contained on them in areas where it is impossible to expect reasonably accurate information in a form format. In the first case spot check interviews are conducted with individuals filling out the forms to check that they have given correct information as interpreted by an independent observer. This would include agreement about such things as the cause of an error or at what point in the development process the error was caused or detected.

In the second case, interviews will be held to gather information in depth on several management decisions, e.g., why a particular personnel organization was chosen, why a particular set of people was picked, etc. These are the kinds of questions that often require discussion rather than a simple answer on a form.

C. Automatic Data Collection: The easiest and most accurate way to gather information is through an automated system. Throughout the history of the project, more and more emphasis will be placed on the automatic collection of data as we become more aware what data we want to collect, i.e., what data is the most valuable and what data we can or need to get, etc. More energy will be expended in the development or procurement of automatic collection tools as the laboratory continues.

The most basic information gathering device is the program development library. The librarian will automatically record data and alleviate the clerical burden from the manager and the programmers. Copies of the current state of affairs of the development library will be periodically archived to preserve the history of the developing product.

A second technique for gathering data automatically is to analyze the product itself, gathering information about its structure using a program analyzer system. A set of modifications to the FACES system is currently underway and will progress as the laboratory gains more experience. These modifications are geared at getting more of the kind of information about the product required for our measures.

D. Database analysis: The above data collected on

the project will be stored in a computerized database. Data analysis routines are being written to collect derived data from the raw data in the database. The data that is being collected is being processed by a PDP11-based system. For ease of implementation, it utilizes the INGRES relational database system [HELD, 75] which runs under the UNIX operating system.

V. Current Status

Beginning in November, 1976, most new software tasks that were assigned by the Systems Development Section of NASA/GSFC were given the added responsibility of filling out the forms, and thus entered our set of screening experiments. At the present time, about a dozen projects are currently involved. These projects are mostly ground support routines to various spacecraft projects. This consists of attitude orbit determinations, telemetry decommutation and other control functions. The software that is produced generally takes from six months to two years to produce, is written by three to six programmers most of whom are working on several such projects simultaneously, and consists of six man-months to ten man-years of effort. Projects are managed by NASA/GSFC employees and the personnel are either NASA personnel or outside contractors.

In June of 1977, the first of the controlled experiments began. Two teams (0 and 1) are assigned tasks to be designed and developed for delivery to the Systems Development Section. The format of these tasks satisfy the experimental design outlined in Section II.

i.e., A_0 XB_0 C_0

A_1 YB_1 C_1

where A_i, B_i, and C_i, represent tasks to be developed by team i and X and Y are training sessions. These tasks will be developed on the PDP-11/70 at NASA/GSFC. One team will consist of in-house NASA/GSFC personnel while the other will consist of contractor personnel. The tasks will consist of five separate subtasks with two comprising project 'A', one project 'B', and two comprising project 'C'. All subtasks require somewhere on the order of three man-months of effort.

Team 1 will be given a training session (Y) after completing the A projects, consisting of several techniques: PDL, Structured Programming, Walk-throughs, use of Librarians, Code Reading, and will also be given a small project B to take into account the necessary learning

curve before Project C is undertaken. Team 0 will also be given a training session and a B project, but will not be taught the above techniques.

For this first controlled experiment, there is complete control of the development process. The A projects enable us to determine the background of the personnel and the C projects enable us to determine the effects of the training sessions. The small B task enables us to filter out much of the learning curve involved in learning new techniques. Due to cost considerations, the duplicate developments must necessarily be kept small; however, the projects are large enough to require team interaction among the programmers and therefore we believe that they are generalizable to larger projects. In addition, the techniques taught in the Y training session are those most applicable to team situations.

A second, longer range, controlled experiment was begun in March, 1977. In this case, several similar large scale projects are being carefully monitored. These projects can be summarized by the following table:

Project	Man Years	Techniques Used
1	6	NONE
2	4½	Structured code, Librarian, code reading
3	4½	Training session Y of experiment 1
4	6	Not yet defined

In this case we are performing C-like experiments of controlled task 1. Due to budgetary restrictions, it is not possible to duplicate the development of each, however, the tasks are highly similar and should give us results similar to the strictly monitored controlled task 1. While we realize that we have less control over this experiment, this controlled experiment allows us to study larger projects. By varying the methodology, we expect to observe differences in project progress.

The next stop will be to define controlled experiment 3, based upon the preliminary results of experiments 1 and 2. It is expected that controlled experiment 3 will begin in early 1978. In this case, the techniques taught in training sessions X and Y and used in C, may be changed to reflect the new techniques to be measured. It is expected that as this process continues over several iterations, quantitative data on various products and development processes will result.

ACKNOWLEDGMENTS

The development of this laboratory has involved the efforts of many people, including Robert W. Reiter, David L. Weiss, Howard J. Larsen, Charles L. Wolf, Frank McGarry, Richard des Jardins, Walter Truszkowski, Robert Nelson, and Keiji Tasaki.

REFERENCES

[AMOR, 73] Amory, W., J. A. Clapp, A Software Error Classification Methodology, MTR 2648, Vol. VII, The Mitre Corporation, June, 1973.

[BAKE, 75] Baker, F. T., Structured Programming in a Production Programming Environment. International Conference on Reliable Software, Los Angeles, April, 1975, (Sigplan Notices 10, 6, June 1, 1975, pp. 172-185).

[BASI, 75] Basili, V. R., A. J. Turner, Iterative enhancement: a practical technique for software development, IEEE Transactions on Software Engineering, I, No. 4, December, 1975, pp. 390-396.

[BASI, 77] Basili, Victor R., Zelkowitz, Marvin J., et al., The Software Engineering Laboratory, University of Maryland Computer Science Technical Report, TR-535, May, 1977, 104 pages.

[BAUM, 63] Baumgartner, J. S., Project Management, Richard D. Irwin, Inc., 1963.

[BOEH, 75] Boehm, B. W., R. K. McClean, D. B. Urfrig, Some Experience Aids to the Design of Large Scale Reliable Software, IEEE Transactions on Software Engineering 1, No. 1, March, 1975, pp. 125-133.

[BROW, 73] Brown, J. R., A. J. De Salvia, D. E. Heine, J. G. Purdy, Automated software assurance, Program Test Methods, Prentice Hall, 1973, pp. 181-203.

[CAMP, 66] Campbell, D. T., J. C. Stanley, Experimental and quasi-experimental designs for research, Chicago, Rand McNally Publishing Co., 1966.

[DAHL, 72] Dahl, O., E. Dijkstra, C. A. R. Hoare, Structured Programming, New York, Academic Press, 1972.

[DOMO, 75] Domonic User Guide, Advanced Technology Group, Data Processing Center, Texas A&M University, 1975.

[ENDR, 75] Endres, A. B., An Analysis of Errors and Their Causes in System Programs, IEEE Transactions on Software Engineering 1, No. 2, June, 1975, pp. 140-149.

[GANN, 75] Gannon, J. D., J. J. Horning, Language Design for Programming Reliability, IEEE Transactions on Software Engineering 1, No. 2, June, 1975, pp. 179-191.

[HAMI, 76] Hamilton, M., S. Zeldin, Higher Order Software - A Methodology for Defining Software, IEEE Transactions on Software Engineering 2, No. 1, March, 1976, pp. 9-32.

[HELD, 75] Held, G., M. Stonebraker, E. Wong, INGRES - A relational data base system, National Computer Conference, 1975, pp. 409-416.

[HELL, 72] Hellerman, L., A Measure of Computational Work, IEEE Transactions on Computers 21, No.5 1972, pp. 439-446.

[MILL, 72] Mills, H. D., Mathematical Foundations for Structured Programming, FSC 72-6012, IBM Corporation, Gaithersburg, Maryland 20760, February, 1972.

[MILL, 75] Miller, E. F., Jr., Methodology for Comprehensive Software Testing, Interim Report, Rome Air Development Center, RADC-TR-75-161, June, 1975, AD# A013111.

[MYER, 75] Myers, G., Software Reliability Through Composite Design, New York, Mason Charter, 1975.

[RAMA, 74] Ramamoorthy, C. V., S. F. Ho, FORTRAN automatic code evaluation system (FACES), part I. Memorandum No. ERL-M-466, Electronics Research Laboratory, University of California, Berkeley, August, 1974.

[RAME, 75] Ramamoorthy, C. V., S. B. F. Ho, Testing Large Software with Automated Software Evaluation Systems, IEEE Transactions on Software 1, No. 1, March, 1975, pp. 46-58.

[REIF, 75] Reifer, D. J., "Automated Aids for Reliable Software," An Invited Tutorial at the 1975 International Conference on Reliable Software, 21-23 April 1975.

[SHOO, 75] Shooman, M. L., M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," Proceedings 1975 Conference on Reliable Software, April 21-23, 1975, pp. 347-362.

[STAY, 76] Stay, J. F., HIPO and integrated program design, IBM Systems Journal 15, No. 2, 1976, pp. 143-154.

[SULL, 73] Sullivan, J. E., Measuring the complexity of computer software, Mitre Corp. Report MTR-2648, Vol. V, June, 1973.

[TAUS, 76] Tausworthe, R. C., Standard Development of Computer Software, Part 1 Methods, Jet Propulsion Lab, Calif. Inst. of Technology, Pasadena, Calif., July, 1976.

[TEIC, 77] Teichroew, D., E. Hershy, PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems, IEEE Transactions Software Engineering 3, No. 1, January, 1977, pp. 41-48.

[THAY, 76] Thayer, T. et al., Software reliability study, TRW Defense and Space Systems Group, National Technical Information Services AD-A030-798, August, 1976.

[VANE, 70] Van Emden, M. H., The hierarchial decomposition of complexity, Machine Intelligence 5, 1970, pp. 361-380.

[VANL, 76] Van Leer, P., Top-down development using a program design language, IBM Systems Journal 15, No. 2, 1976, pp. 155-170.

[WALS, 77] Walston, C. E., C. P. Felix, A method of programming measurment and estimation, IBM Systems Journal, No. 1, 1977, pp. 54-73.

[WOLV, 72] Wolverton, R. W., The Cost of Developing Large Scale Software, TRW Software Series TRW-SS-73-01, March, 1972.

Research supported in part by grant NSG-5123 from the National Aeronautics and Space Administration to the University of Maryland.

P.4 D2-61 80018

OPERATION OF THE SOFTWARE ENGINEERING LABORATORY*

Victor R. Basili and Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

The paper discusses the current status of the Software Engineering Laboratory. Data is being collected and processed during the development of several NASA/Goddard Space Flight Center ground support projects. The data is used to evaluate software development disciplines and various models and measures of the software development process. Emphasis is placed upon models of resource estimation, the analysis of error and change data, and program complexity measures.

- - - - -

The Software Engineering Laboratory is a research project between NASA/Goddard Space Flight Center and the Department of Computer Science of the University of Maryland. Ground support software, in the six to twelve man-year range, developed for the Systems Development Section of NASA, is studied in detail for determining the dynamics of software development and the effects of various features and methodologies on this development [Basili and Zelkowitz 77]. Most data is collected in a set of reporting forms that are either filled out periodically by all project personnel (e.g., a weekly Component Status report) or whenever certain events occur (e.g., a Change Report Form when an error is corrected). This report describes the activities of the laboratory for the last twelve months.

The initial goal of the Software Engineering Laboratory was the collection of valid data and the entering of this data into a computerized data base. During the last twelve months, this process has been implemented and the analysis of the data has begun. This report will be divided into four sections briefly outlining each of the major activities undertaken by the laboratory: (1) Data Collection Activities, (2) Resource Estimation, (3) Error Analysis, and (4) Program Complexity.

Data Collection Activities

The first task of the laboratory was to implement a data base that accurately reflected software development. The INGRES data base system operating under the UNIX operating system on a PDP 11/45 computer at the University of Maryland was chosen as the basic data base system [Stonebraker 76]. This activity resulted in the following steps:

A generalized table-driven program was implemented that converted the raw typed-in forms to a format acceptable to INGRES. However, it soon became apparent that the major problems were not program oriented, but were in the human communication necessary to carry out this activity.

Forms were frequently filled out containing names not yet recognized by the data base. Other fields were sometimes missing or unclear. Constant interaction between the University personnel and the programmers filling out the forms became necessary in order to solve this problem.

Thus, the first change in procedure was to rewrite the data validation program for the PDP 11/70 at NASA. Forms are turned in to a single individual assigned to the Laboratory. The form is scanned manually and any errors are brought to the attention of the programmers. The validation program finds additional errors that can be quickly corrected. Correct forms are written to tape for transmittal to the University.

This activity led to a second task--a revision of the forms. We observed that the programmers preferred a "checklist" format rather than a set of "fill in the blanks," even if more checks were needed than blanks. Many of the early forms were studied for typical responses and the forms were modified appropriately. In addition, some seemingly useful information, but based upon data that was generally not being given by the programmers, has been deleted in order to lessen the apparent overhead perceived by the programmers participating in the laboratory.

Another activity now under way is the movement of the data base to the PDP 11/70 at NASA. Due to the smaller size of the PDP 11/45 at the University and the relative inefficiency of INGRES for large-scale applications, operation of the University setup is starting to become cumbersome. The PDP 11/70 should eliminate that problem.

Summarizing the activities of the past year, several schemes were developed and we now have evolved a semi-automatic process for entering data into a data base:

*Research supported in part by grant NSG-5123 from NASA/Goddard Space Flight Center to the University of Maryland.

1. Forms are turned in and manually scanned for errors.
2. The forms are entered into a validation program at NASA. If errors are present, the form is returned for corrections. If correct, it is written to tape.
3. The tape of correct forms is brought to the University for data base entry.

(By January 1979, it is expected that the corrected tape will also be entered into a data base on NASA's PDP 11. At that time the decision will be made as to whether to keep the University data base or to interface with NASA's.)

Resource Estimation

One early research activity was the investigation of resource utilization. The Rayleigh curve has been studied for larger projects and the applicability of this theory in the smaller NASA environment was investigated.

Cumulative costs for large-scale software development has been shown to approximate the curve $K\ (1 - e^{-a\ t^2})$ where K is the total project cost and t is the elapsed time since project initialization [Putnam 76]. This is usually represented in its differential form called a Rayleigh curve: $(2\ K\ a\ t\ e^{-a\ t^2})$, and represents the rate of consuming resources. This curve looks somewhat like a normal distribution with a more extended tail (see Figure 2).

In our NASA environment, from the general project summary form, these numbers are obtained:

1. K_e, total estimated cost of the project in hours of effort. Counting overhead items, like typing support and librarians, total costs (K) are usually 112% of K_e.
2. Y_d, the maximal effort per week. From this, constant a can be developed, $a = (1/2y_d^2)$.
3. T_a, the estimated date of acceptance testing. In NASA's environment this usually occurs after 88% of total expenses are consumed.

Since the Rayleigh curve has two parameters (K and a) and the general project summary gives three (ie, y_d and t_a), the applicability of the Rayleigh curve to this environment can be checked by using two of these estimates to predict the third.

Figure 1 represents this analysis for two projects. Figure 1.A presents the estimated data from the general project summary. In Figure 1.B, t_a was estimated from K_e and y_d and y_d was estimated from K_e and t_a. Finally, Figure 1.C presents the actual data.

Figure 2 plots some of this for these two projects. While Figure 1 shows that K_e and y_d are accurate predictors of t_a (e.g., an estimate of 60 weeks for project A, only a two-week error from the actual 62 weeks, and a much better estimate than the initial estimate of 46 weeks), the plots of this curve differ from actual resource consumption. The conclusion seems to be that the Rayleigh curve is only a crude approximation to reasonable consumption. (See [Basili and Zelkowitz 78a] for more details.)

In order to test this further, several other curves were correlated with the actual data (parabola, trapezoid and straight line) [Mapp 78]. All had as good correlations to the data as the Rayleigh curve. Thus, the Rayleigh curve was no better, and in many cases worse, than other estimates.

In addition, Norden's original assumptions involve a linear growth in the rate of understanding a project [Norden 70]. In reality, this learning curve slows as personnel become familiar with a project. Based upon this assumption, [Parr 78] has developed a curve based upon the hyperbolic secant that may be more applicable in the NASA environment. This and other theories related to the Rayleigh curve are now being studied.

The evaluation performed in [Basili and Zelkowitz 78a] has led to a set of procedures that can be used to monitor project development in a production environment. While the full set of seven reporting forms may prove to be too much overhead, a set of procedures using only three forms can be used to monitor project progress with reasonable accuracy [Basili and Zelkowitz 78b]: the General Project Summary, submitted at each project milestone; the Resource Summary, giving hours worked by all project personnel by week; and the Change Report Form giving all changes to the system.

Error Analysis

The principal motivations for studying errors and changes have been to discover the effects of various factors on the number and kinds of errors made in system developments, and to find ways to evaluate proposed software development methodologies.

To study this, a number of tasks have been performed. First, to assure that all of the forms have been filled out in a consistent manner, a glossary of terms has been defined and made available to the participants of the monitored software development process. Second, a set of questions of interest were defined which were used to motivate both the form content and organize the kinds of data required in the form of interviews with the participants. Questions of interest include the following:

What are good ways of characterizing error-proneness of software development? Measures such as the total number of errors, errors per line of code, errors per man hour, errors per component type where type refers to the kind of sub-application or level of complexity, number of fixes per project phase per component are being considered. We are also looking at relationships between the various types of error classification.

What are the major sources of errors? One possible characterization is by analyzing whether errors are traceable back to requirements, specification, interface definition or intra-component design, or clerical activities or the hardware environment.

What are appropriate ways of measuring ease of software change? Data is being collected on effort per change in terms of time, the number of fixes required for the change, and the number of errors generated by the change.

What is the effect of continual change on a software product? Data is gathered on the cost of change as a function of time and cumulative changes.

What type of changes cause most of the errors? This may be very environment dependent or it may give some insights into improved organizations and methodologies for software development.

What is the effect of personnel organization on errors? Data is being collected on correctness as measured by errors per number of people working on a piece of software. Again, this should shed some insights on the way to organize tasks within a given environment.

What types of changes predominate during software development? Knowing this should aid in designing software to anticipate the possible changes.

What are the most prevalent error detection and correction techniques? Knowing what is used most often and what works and at what cost will help in determining what should be used for what classes of errors in what environment.

What is the effect of various constraints, such as time and memory on error distributions? Understanding this will permit better evaluation of the tradeoffs in software management.

These are but some of the types of questions the error analysis phases of the Software Engineering Laboratory is studying. The data for most of these questions is gathered from the Change Report Form, with additional information from the other forms and follow-up interviews to validate the accuracy of the information and gather additional data not easily collected in a form format.

Based upon the above questions, several "first order metrics" have been defined and software has been developed to gather information from the data base. Data is being gathered at a slow pace partly because of the current backlog of Change Report Forms which have not yet been entered into the data base, and partly because of the refinement of the form as mentioned in the section on Data Collection Activities. Early analyses on a couple of projects, however, do indicate that the distribution of errors during development appears to approximate the Rayleigh curve as found by [Schick and Wolverton 78].

Continued effort will deal with the gathering of information to answer the basic questions of interest, further development of new questions of interest, and possible "second order metrics" based on the intuition gathered from the current studies.

Program Complexity

There is much interest in measures of complexity of the software product, the valid aspects of the product that effect human understanding. There is an interest in quantitatively measuring these aspects so that characteristics of programs that make them more or less error prone, harder to modify, or more difficult to develop can be better understood and recognized. Measures proposed in the literature may even be used to characterize differences in the development process.

Work has been done at the University of Maryland to analyze and compare the development of software in an experimental environment to determine the effects of development methodologies [Basili and Reiter 78]. The experiment involved the use of three different types of development: Single individuals using ad hoc techniques, groups of three using ad hoc techniques, and groups of three using a structured programming methodology. Results have shown that there is some distinction in the product using very rough measures of the program characteristics, such as number of if statements, number of globals, etc. Based on this study, the organized group lies somewhere between the ad hoc group and the single individual. However, with regard to process measures, the organized group has shown less computer runs in all phases of development and less errors (using a measure of errors called program changes which is algorithmically computable based on different versions of the software product [Dunsmore 78]).

It is planned to implement the promising measures from this research on programs from the NASA environment. Versions of the systems developed at NASA have been saved and will be compared for program changes and checked against the result from the Error Report in the Change Report Forms. Further work is being done in automating and comparing various complexity measures. These include several of our own measures (prime program hierarchy, data bindings, etc.) as well as some of the measures that have appeared in the literature [Halstead 77; McCabe 76].

References

[Basili and Zelkowitz 78a] Basili, V. and M. Zelkowitz, Analyzing Medium-Scale Software Development, Third International Conference on Software Engineering, Atlanta, Georgia, May 1978, pp. 116-123.

[Basili and Zelkowitz 78b] Basili, V. and M. Zelkowitz, Measuring Software Development Characteristics in the Local Environment, Journal of Computers and Structures, 1978, 5 pp. (to appear).

[Basili and Zelkowitz 77] Basili, V. and M. Zelkowitz, The Software Engineering Laboratory: Objectives, ACM SIGCPR Annual Conference, Washington, D. C., August 1977, pp. 256-269.

[Basili and Reiter 78] Basili, V. and Reiter, R. An Experimental Comparison of Software Development Approaches, University of Maryland, Computer Science Technical Report TR-688, August 1978.

[Dunsmore 77] Dunsmore, H. E. and Gannon, J. D., Experimental Investigation of Programming Complexity, Proceedings of the 16th Annual Technical Symposium: Systems and Software Washington, D. C. (June 77) pp. 117-125.

[Halstead 77] Halstead, M., Elements of Software Science, Elsevier Computer Science Library 77.

[McCabe 76] McCabe, Thomas J., A Complexity Measure, Transactions on Software Engineering, Dec. 76, Vol. SE-2, No. 4, pp. 308-320.

[Mapp] Mapp, T., Applicability of the Rayleigh Curve to the SEL Environment, University of Maryland, Department of Computer Science, Scholarly Paper, May 1978.

[Norden 70] Norden, P., Use Tools for Project Management, Management of Production, M. K. Starr (ed), Penguin Books, Baltimore, Maryland, 1970, pp. 71-101.

[Parr 78] Parr, F., An Alternative to the Rayleigh Curve Model for Software Development Effort (submitted for publication).

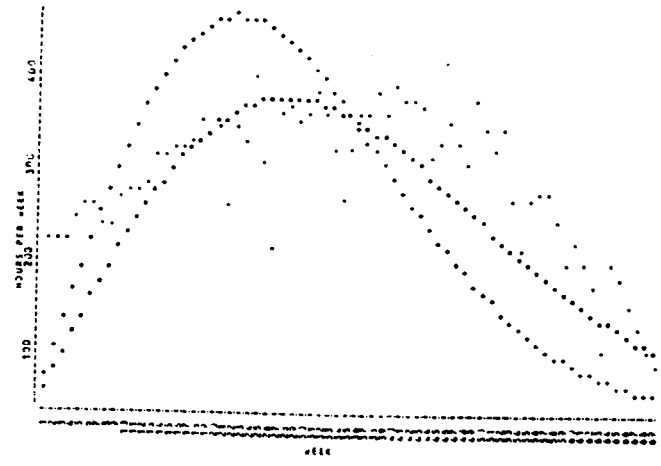
[Putnam 76] Putnam, L., A Macro-estimating Methodology for Software Development, IEEE Computer Society Compcon, Washington, D. C., September 1976, pp. 138-143.

[Schick and Wolverton 78] Schick, George J. and Wolverton, Ray W., An Analysis of Competing Software Reliability Models, IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, March 1978, pp. 104-120.

[Stonebraker 76] Stonebraker, M., E. Wong and P. Kreps, The Design and Implementation of INGRES, ACM Transactions on Data Base Systems 1, No. 3, 1976, pp. 189-222.

	PROJECT A	PROJECT B
A. Initial Estimates from General Project Summary		
Ka, Resources needed (hrs)	14,213	12,997
Ta, Time to completion (wks)	46	41
Yd, Maximum resources/wk (hrs)	350	320
B. Completion Estimates Using Rayleigh Curve		
K, Resources needed (hrs)	16,151	14,770
Est. Yd with Ta fixed (hrs)	440	456
Est. Ta with Yd fixed (hrs)	58	58
C. Actual Project Data		
K, Resources needed (hrs)	17,741	16,543
Yd, Maximum resources (hrs)	371	462
Ta, Completion time (wks)	62	54
Ta, Estimated using actual values of K and Yd (wks)	60	43

Figure 1: Estimating Ta and Yd from General Project Summary Data



* - Estimating curve with Yd (maximum resources) fixed
\+ - Estimating curve with Ta (completion date) fixed
. - Actual data

Figure 2. Estimated resource expenditures curve

SECTION 3 – RESOURCE MODELS

OMIT

SECTION 3 - RESOURCE MODELS

The technical papers included in this section were originally published as indicated below:

- Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science, New York: Computer Societies Press, 1979, copyright 1979 IEEE (reprinted by permission of the publisher)
- Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering, New York: Computer Societies Press, 1981, copyright 1981 IEEE (reprinted by permission of the publisher)
- Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," Journal of Systems and Software, February 1981, vol. 2, no. 1, copyright 1981 Elsevier-North Holland (reprinted by permission of the publisher)

R.6 D3-61 80019

RESOURCE ESTIMATION FOR MEDIUM-SCALE SOFTWARE PROJECTS*

Marvin V. Zelkowitz+

Department of Computer Science
University of Maryland
College Park, Maryland 20742

The ability to forecast accurately costs and development times for software development projects is an important management tool. A theory for such estimation on large scale developments has been proposed by Norden and refined by Putnam, and is based upon a statistical model which yields a Rayleigh curve as the best estimate of software costs and times.

The Software Engineering Laboratory has been established at the University of Maryland and NASA Goddard Space Flight Center for studying the mechanics of medium-scale development. This paper will describe the Laboratory, and will explain some of the research that is investigating the Norden-Putnam model in the NASA environment.

The ability to accurately forecast the resource needs in developing software is an important management criteria. Underestimating those needs could lead to late product delivery or even to project failure. Overestimating those needs can lead to wasted resources with no guarantee that the project can be completed in less time, as compensation.

This report will be divided into three sections. Part 1 will be a general description of resource estimation for software development. In Part 2, a particular methodology, based upon the research of Norden and Putnam will be described. Finally, Part 3 will describe the Software Engineering Laboratory of the University of Maryland, and will describe some of the research relevant to the issues of resource estimation.

1.0 RESOURCE ESTIMATION

Management of a software development typically has 2 major resources to control - people and computer usage. Controlling the personnel will usually regulate the use of the computer. Thus resource estimation typically reduces to controlling the number of people assigned to a given project. The problem is to define a way to control (or estimate) this size.

Assume via some method (to be described later) that a project will require 96 man-months of effort. The manpower loading curve of Figure 1 will be a typical description of this effort. However, this figure leads to the conclusion that Figures 2 and 3 are equally valid. However, we know that this is not the case. Personnel cannot be traded for months [Brooks75], and resource estimation consists of more than simply deciding whether to have more personnel with an earlier delivery date or fewer personnel with a later delivery date.

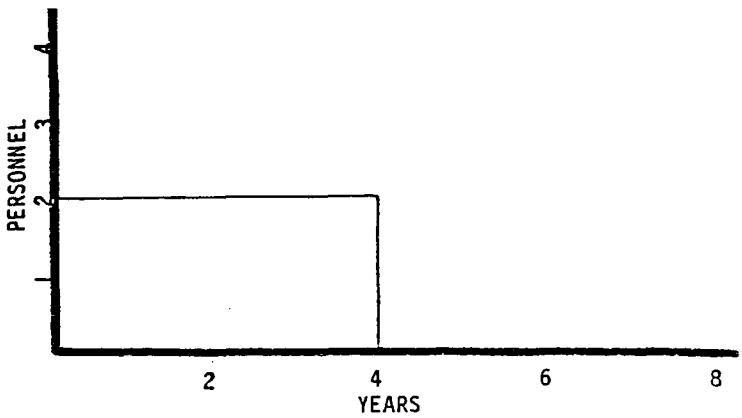


Figure 1. Typical Resource Usage

---

* - Research supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland.

+ - Also with the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC. 20234

Before describing a general theory of resource estimation, a general estimation methodology will be described. How does one estimate resources for software? In order to answer this, other engineering fields can be looked at as models of the process.

Engineers have developed a relatively standard approach towards resource estimation [Gallagher65]. One such approach includes the following steps:

1. Develop an outline of the requirements.
2. Gather similar information, such as data from similar projects.
3. Select the basic relevant data.
4. Develop estimates.
5. Do final evaluation.

In order to develop these estimates (step 4), the steps to be followed includes:

4(a). Compare the project to previous similar projects.
4(b). Divide the project into units, and compare each unit with other similar units.
4(c). Schedule work by month, and estimate resources by month.
4(d). Develop standards that can be applied to work.

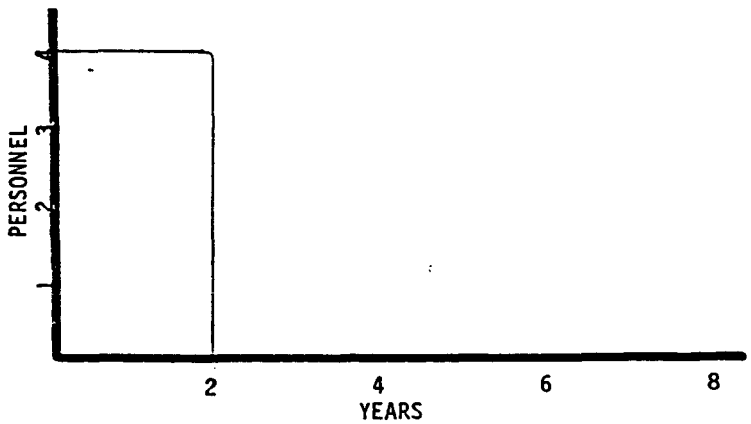


Figure 2. Assumed Resource Usage

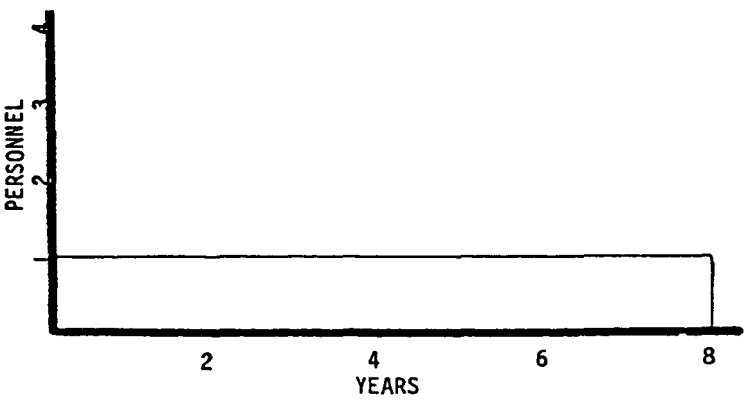


Figure 3. Assumed Resource Usage (cont.)

How are estimates generally made? Three techniques are generally used:
(1) Expert judgment An educated "guess" by the chief designers.
(2) Algorithmic analysis Use of an explicit algorithm, if known.
(3) Top down To divide the project into units in a hierarchical manner.

In the software field, we have trouble at almost every step in the process. What do we mean by requirements? At a recent Computer Society conference entitled Specifications of Reliable Software [SRS79], there was no firm agreement as to how a requirement or specification should be described. We do not have any concept like the engineering blueprint to apply to developments (although the term has certainly been used).

We also have little background data to draw on (step 2). While civil engineers have been building bridges for thousands of years, software is only 30 years old.

What is a "unit" of a software system (Step 4(b))? We do not have any firm idea of what a standard module, component or subroutine of a system should be. In the case of the certain mathematical functions (e. g., sin, log, square root) the problem is relatively easy; but little software can be broken down into such easily described functions. While such concepts as "levels of abstraction" and "modular programming" are attempts at answering this need, there is no effective definition that can be used efficiently in producing software.

The problems of software standards is another problem area. In the construction industry, building codes dictate how iron, steel or glass are to be used. For example, if a beam is estimated to need to support a weight of 10,000 kilograms, then a beam capable of supporting 20,000 kilograms may be used for reliability. However, how do you make a software subroutine "twice as reliable?" What do we even mean by "software reliability?"

While the situation has so far been painted as very bleak, in reality, we are not that bad off. First of all, most engineering fields fail badly when applied to new technology. For example, the Alaskan Oil Pipeline was estimted to cost $900 million, yet was completed at a cost of over $9 billion. Similarly, although most buildings are quite sturdy, some do collapse during construction. In software, we just do not have the background to build on - although that experience is growing.

Having just purchased a new house, I have come to realize another distinction between engineering and software. The builder of this house has recently constructed about 30 others of the same model. However, each differs in some significant way. But there is a certain degree of robustness in the design to allow all of them to be functional and essentially the same. This concept of robustness is missing from most software designs. Also, the builder claims that the house is constructed from "over 4000 parts". This is actually a small number when compared to over 100,000 instructions for a typical large program. A software system may actually be a very complex system, and the fact that reliability and robustness are hard may not be really surprising.

To help in estimating needed resources, it is now recognized that software passes through several distinct stages during its lifetime. This has been called the software life cycle. The effort required for each stage is approximately as follows [Zelkowitz79]:

Requirements - 10%
Specification - 10%
Design - 15%
Code - 20%
Module testing - 25%
Integration testing - 20%

It is now recognized that the maintenance stage takes a significant part of the effort - ranging up to 70% of the total development and maintenance costs. Using these figures as a guideline, management can monitor progress and estimate projected costs.

2.0 RAYLEIGH CURVE ESTIMATION

Effective estimation techniques are being developed by applying results from computer hardware reliability theory [Putnam77]. The cumulative expenditures over time for large scale projects (over 50 man-years of effort) has been found to agree closely with the following equation:

$$E(t) = K(1-\exp(-at^2))$$

where E(t) is the total amount spent up to time t, K is the total project cost, and a is a measure of the shape of the expenditure curve. This relationship is usually expressed as a differential equation, called a Rayleigh curve:

$$E'(t) = 2Kat\exp(-at^2)$$

where E'(t) is the rate of expenditures, or the amount spent on the project during time unit t (Figure 4).

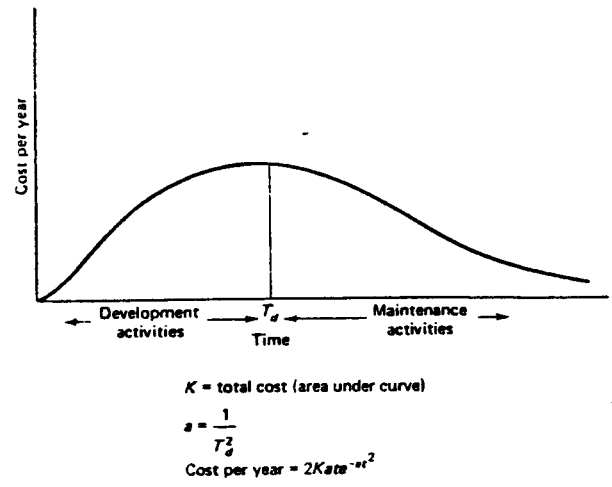


Figure 4. Rayleigh Curve

The theory of the Rayleigh curve is based upon the following assumptions:

1. The number of problems to solve in building a software product is finite, but of an unknown number.
2. The process of information gathering, thinking about possible solutions, and identifying alternatives all consume time. These design decisions convert one of the unsolved problems into a solved problem.
3. The occurrence of these events of converting unsolved into solved problems is independent and random.

This leads to Poisson solution with an exponential interevent arrival time satisfying the equation:

$$y = \exp(-lt)$$

where y is the probability of the problem remaining unsolved by time t.

4. The number of people working in a group at any time is proportional to the number of problems "ripe" for a solution. This assumes that each individual is working independently on unsolved problems to solve.

If we let

Pr(T>t)= probability that no event occurs in the interval [0,t]

then from the Poisson assumption:

$$Pr(T>t)=\exp(-lt)$$

Since $Pr(T\leq t)+Pr(T>t)=1$, then the probability of an event occurring in [0,t] is just:

$$Pr(T<t) = 1-Pr(T>t)$$
$$= 1-\exp(-lt)$$

The rate of solving problems is the derivative, or:

$$f(t) = -l \exp(-lt)$$

3'. If we now assume that after an event occurs, p is the probability that the event is actually solved (e. g., the correct decision was made). This leads to:

$$Pr(T<t) = 1-\exp(-plt)$$
$$f(t) = -pl \exp(-plt)$$

If we further assume that the probability of success (p) is a function of time (p(t)), then the solution is:

$$pr(T>t) = \exp(-l \int_0^t P(J)\, dJ)$$

$$Pr(T<t) = 1 - \exp(-l \int_0^t P(J) dJ)$$

$$f(t) = l\, p(t) \exp(- \int_0^t l\, p(J)\, dJ)$$

Based upon empirical data, the best fit for p(t) is bt, yielding the formulas (after substituting a for lb/2, and multiplying by K, the cost of a project):

$$Pr(T<t) = K(1-\exp(-at^{**}2))$$
$$f(t) = 2\,Kat \exp(-at^{**}2)$$

The following argument can be used to justify the feasibility of this development. If we let y(t) represent the total expenditures on a project, then the rate of spending can be rewritten as:

$$y'(t) = 2Kat(K-y)$$

This equation contains 2 non-constant terms: t and (K-y). As a project moves towards completion, t increases so the rate of progress increases. This is due to the effects of the "learning curve" as personnel become more familiar with the task. Working against this trend is the term (K-y) which decreases as y increases. This is due to the project becoming more complex as the project nears completion.

The Rayleigh curve contains two parameters, K and a; however, there are three general characteristics that can be used to measure a project: total cost, rate of development and completion date. Two of these can be used to measure the third. This technique has been investigated, and will be discussed in the next section.

3.0 SOFTWARE ENGINEERING LABORATORY

The Software Engineering Laboratory, a joint research project between NASA Goddard Space Flight Center and the Department of Computer Science of the University of Maryland, was established in August, 1976 to study program development in the NASA environment. The goals of the laboratory are to determine the dynamics of program development, and to recommend techniques to produce cost effective software. These goals can be broken down into the following three major tasks:

1. A reporting mechanism for monitoring project progress was developed. A set of 7 forms were developed, and each project that is being monitored is required to fill out each form periodically.

2. Data (from the forms) is being collected and stored on a computerized data base. About 1.2 millions characters of data from 4000 forms are now in the data base, with another 4000 forms now being processed. This represents data from about 25 projects, varying in size from single programmer tasks lasting about a month, to 100 man-month efforts requiring 10 individuals about a year to complete.

3. The data from the various forms are being analyzed from various perspectives. The issues now under study include: Programming errors and reliability, Productivity measures, Complexity measures on the finished software product, and Resource estimation measures.

The forms that are being collected include the following:

General Project Summary. This form is filled out at each major project milestone. This is approximately six times per project, or once every 6 to 8 weeks.

Component Summary. For each component of a system (e.g., subroutine), a component summary form is filled out at least twice - once when the component is design, and once when it is completed. This form is similar to the general project summary, but is for a smaller piece of the system.

Component Status Report. This form is filled out weekly by all project personnel, and is the main form used in keeping track of progress. This form lists, for each programmer, the time spent on each component, and the activity involved with that component (e. g., design, code, test).

Resource Summary. This is a summary accounting, by week, of the total number of

hours spent on the project by all project personnel.

Change Report Form. This form is used to report any change or error made during development.

Computer Run Analysis. An entry is made on this form each time a computer run is made. It briefly describes the purpose and results of the run.

Attitude System Maintenance. A form is filled out during the operational phase of a project whenever the source programs need to be modified.

Note: Not every form is used for every project; however, this is the complete list of the forms that are used.

Figure 5 presents a summary of the data collected from 15 different projects. As shown, the size of the programs varied from 2,000 to 112,000 statements, and required from 1 to 11 people to complete. Most of the projects consisted of a variety of tasks (e. g., scientific computing, utility programs, data processing, etc.), and used a variety of techniques.

The remainder of this report will discuss some of the issues in resource estimation that are under investigation by the Laboratory. Additional information can be found in [Basili78].

As discussed earlier, the Rayleigh curve has two parameters, K and a, and a system can be described by three general characteristics:

(1) Its total cost
(2) Its rate of development
(3) Its completion date

But two of these characteristics are enough to determine the constants K and a. When a project is initiated, the proposed budget is an estimate of K and the available personnel permits a to be calculated. Assuming that requirements analysis determines that these figures represent an accurate assessment of the complexity of the problem, then the estimated completion date can be computed, and must not be set arbitrarily during the specifications stage of development.

This technique was tried with data collected by the Laboratory. The results are summarized by the following table:

SOURCE STATEMENTS(K)	MACHINE	MAX PEOPLE	TYPE SOFTWARE	METHODOLOGY	EXTRACTED DATA
2	PDP-11	1	1-5	2	•••
60	S/360	8	1-5	5-6-7	•
40	S/360	10	1-4-5	3-4-8	•
112	S/360	10	1-4-5	1-2-5-6-7	•••
55	S/360	11	1-4-5	1	•••
30*	PDP-11	3	2-5	2	••
2*	NSSC-1	2	1-4	6	•
5	S/360	3	1	5-6-7	•
3	S/360	2	1	5-6-7	•
20	PDP-11	3	1-3-5		••
45	S/360	7	1-5	7-3-6	•••
30	PDP-11	3	1-2-3-4-5	8	•••
30	S/360	3	1-2-3-4-5	2-5-6	•••
47	S/360	8	1-5	1-3	•••
70	S/360	6	1-4-5	3-4-5-6-7-8	•••

TYPE SOFTWARE

1. SCIENTIFIC
2. UTILITY
3. DATA PROCESSING
4. REAL TIME
5. GRAPHICS

METHODOLOGY

1. CHIEF PROGRAMMER
2. TOP DOWN
3. PRE-COMPILE-STRUCTURE
4. PDL
5. WALK THROUGHS
6. CODE READING
7. LIBRARIAN
8. FORMAL TEST PLAN (DURING DEVELOPMENT)

EXTRACTED DATA

• SOME GOOD DATA
•• GOOD DATA
••• VERY GOOD DATA

* ASSEMBLER LANGUAGE(ALL OTHERS FORTRAN)

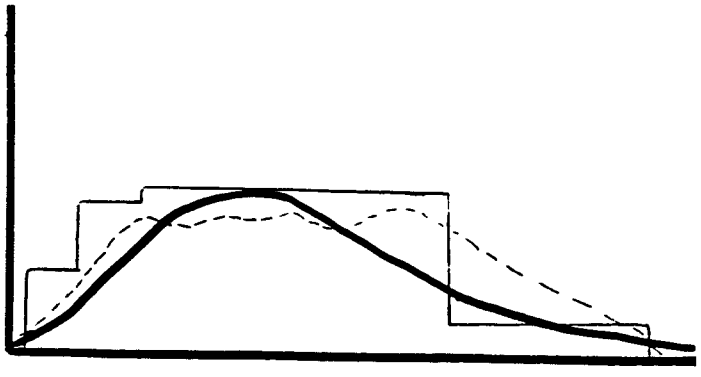
Figure 5. Projects studied by the Software Engineering Laboratory

	PROJECT A	PROJECT B
INITIAL PROJECTIONS		
Resources needed (hrs)	14,213	12,997
Time to complete (wks)	46	41
Max. res./week (hrs)	350	320
RAYLEIGH CURVE ESTIMATES		
Resources needed	16,151	14,770
Time to complete	58	58
ACTUAL PROJECT DATA		
Resources needed	17,742	16,543
Time to complete	62	54

The results turned out to be good, but inconclusive. The Rayleigh curve gave a good estimate for project size and estimated completion date; however, the shape of the curve for the actual data was not good (Figure 6). For large projects, the effects of individual management decisions become "lost" in the "law of large numbers". However, in our smaller projects, such management decisions do affect the loading curve (Figure 6). Current research is investigating alternative models to account for such variances.

4.0 CONCLUSIONS

The Software Engineering Laboratory has been organized to study software development in a production environment. From the experience gained from operation of the laboratory, we can state the following conclusions:



"Ideal" Rayleigh curve

"Assumed" Management curve

----Actual data

Figure 6. Real vs. Theoretical models

(1) Data collection is important. Obtaining accurate data of project development goes a long way towards eliminating much of the folklore of development. How much time is actually spent in design? code? testing? Do newer techniques (e. g., code reading, walkthroughs, design languages) really help? Without objective data, these questions cannot be answered.

(2) Obtaining accurate data is hard. Many programmers view the collection of data as excess overhead. Feedback is needed to convince them that the data is useful. In addition, missing data, and consistency in filling out the forms is of primary importance.

(3) Various models do seem to describe the software development process. This paper has emphasized the resource estimation problem based on the Rayleigh curve. Other research using data from the laboratory is investigating program complexity, reliability, productivity, and error rates.

5.0 REFERENCES

[Basili78] Basili V. and M. Zelkowitz, Analyzng medium-scale software development, International Conference on Software Engineering, Atlanta, Ga, May, 1978, pp. 116-123.

[Brooks75] Brooks F., The Mythical Man Month, Addison Wesley Publishing Co, Reading, Mass., 1975.

[Gallagher65] Gallagher P. F., Project Estimating by Engineering Methods, Hayden Book Co., New York, 1965.

[Putnam77] Putnam L. and R. Wolverton, Quantitative Management: Software Cost Estimating, IEEE Computer Society tutorial, IEEE, New York, November, 1977.

[SRS79] Proceedings of a conference on Specifications of Reliable Software, IEEE Computer Society, New York, 1979.

[Zelkowitz79] Zelkowitz M., A. Shaw, and J. Gannon, Principles of Software Engineering and Design, Prentice Hall, Englewood Cliffs, NJ, 1979.

A META-MODEL FOR SOFTWARE DEVELOPMENT RESOURCE EXPENDITURES

John W. Bailey and Victor R. Basili

Department of Computer Science
University of Maryland, College Park 20742

ABSTRACT

One of the basic goals of software engineering is the establishment of useful models and equations to predict the cost of any given programming project. Many models have been proposed over the last several years, but, because of differences in the data collected, types of projects and environmental factors among software development sites, these models are not transportable and are only valid within the organization where they were developed. This result seems reasonable when one considers that a model developed at a certain environment will only be able to capture the impact of the factors which have a variable effect within that environment. Those factors which are constant at that environment, and therefore do not cause variations in the productivity among projects produced there, may have different or variable effects at another environment.

This paper presents a model-generation process which permits the development of a resource estimation model for any particular organization. The model is based on data collected by that organization which captures its particular environmental factors' and the differences among its particular projects. The process provides the capability of producing a model tailored to the organization which can be expected to be more effective than any model originally developed for another environment. It is demonstrated here using data collected from the Software Engineering Laboratory at the NASA/Goddard Space Flight Center.

INTRODUCTION

Several resource estimation models for a software-producing environment have been reported in the literature [1,2,3,4,5,6,7,8,9], each having been developed in a different environment, each having its particular strengths and weaknesses but with most showing fairly poor characteristics concerning portability to other environments. It is becoming apparent that it is not generally possible for one software development environment to use the algorithms developed at another environment to predict resource consumption. It is necessary for each environment to consider its own past productivity in order to estimate its future productivities. Traditionally, a good manager can estimate resource consumption for a programming project based on his past experience with that particular environment. A model should be able to do the same, and can serve as a useful aid to the manager in this estimating task.

However, if a manager uses a model developed at another environment to help him in his estimations, he will usually find that his intuitive estimates are better than any from the model. It would be advantageous for his software-development organization to generate a model of its own by duplicating the basic steps taken in the development of some outside environment's estimation model. The organization could parallel its own model's development with the development of the existing model, making decisions along the way with respect to which factors have an effect on its software environment, and could mold the newly emerging model to its specific environment. This is seen as an additional advantage over those models which are only "tuned" to the user's environment via a set of specified parameters, since in the latter case there may be no way to express certain peculiarities of the new environment in terms which the model can handle. When one considers in general how poorly a model from one environment fits another environment, it seems that such peculiarities are the rule rather than the exception. Unfortunately, there have been few attempts to reveal the steps taken in generating a resource estimation model which would be helpful to any organization wishing to establish a model for its own use.

This paper is a first attempt by the Software Engineering Laboratory of the University of Maryland at College Park to outline the initial procedures which we have used to establish this type of model for our environment. It is hoped that the framework for the model presented here is general enough to help another software development organization produce a model of its own by following a similar procedure while making decisions which mold the model to its own environment.

One basic approach will be outlined and developed here, but several variations will be discussed. The type of model used is based on earlier work of Walston and Felix at IBM Federal Systems Division and Barry Boehm at TRW in that it attempts to relate project size to effort. Some reasonable measure is used to express the size of a project, such as lines of source code, executable statements, machine instructions or number of modules, and, a base-line equation is used to relate this size to effort. Then, the deviations of the actual projects from this prediction line are explained by some set of factors which attempt to describe the differences among projects in the environment. These factors may include measures of skill and experience of the programming team, use of good programming practices and difficulty of the project.

Several of the alternatives became apparent during our study and these are mentioned when appropriate even if they are not examined further here. Although some of the details and ideas used in this study may not pertain to other environments, it is hoped that enough possibilities are given to show the general idea of how the technique

CH1627-9/81/0000/0107$00.75 © 1981 IEEE

we used can be applied. The study now involves complete data on eighteen projects and sub-projects but was begun when we had complete data on only five projects. It is hoped that the presentation of our work will save other investigators who are developing a model some time or at least provide a point of departure for their own study.

Background

There exist many cost estimation models ranging from highly theoretical ones, such as Putnam's model [1], to empirical ones, such as the Walston and Felix [2] and the Boehm model [3]. An empirical model uses data from previous projects to evaluate the current project and derives the basic formulae from analysis of the particular data base available. A theoretical model, on the other hand, uses formulae based upon global assumptions, such as the rate at which people solve problems, the number of problems available for solution at a given point in time, etc. The work in this paper is empirical and is based predominantly on the work of Walston and Felix, and Barry Boehm.

The Software Engineering Laboratory (SEL) has worked to validate some of the basic relationships proposed by Walston and Felix which dealt with the factors that affect the software development process. One result of their study was an index computed with twenty-nine factors they judged to have a significant effect on their software development environment. As part of their study, they proposed an effort equation which was of the form $E = 5.2*L^{.91}$ where E is the total effort in man-months and L is the size in thousands of lines of delivered source code. Data from SEL was used to show that although the exact equation proposed by Walston and Felix could not be derived, the basic relationship between lines of code and effort could be substantiated by an equation which lay within one standard error of estimate for the IBM equation, and in a justifiable direction [10]. Barry Boehm has proposed a model that uses a similar standard effort equation and adjusts the initial estimates by a set of sixteen multipliers which are selected according to values assigned to their corresponding attributes. In attempting to fit an early version of this model, but with the SEL data, it was found that because of differing environments, a different baseline equation was needed, as well as a different set of environmental parameters or attributes. Many of the attributes found in the TRW environment are already accounted for in the SEL baseline equations, and several of the attributes in the SEL model which accounted for changes in productivity were not accounted for in the Boehm model, presumably because they had little effect in the TRW environment. Based upon this assumption and our experience with the IBM and TRW models, the meta model proposed in this paper was devised.

The SEL Environment

The Software Engineering Laboratory was organized in August, 1976. Beginning in November, 1976, most new software tasks that were assigned by the System Development Section of NASA/Goddard Space Flight Center began submitting data on development progress to our data base. These programs are mostly ground support routines for various spacecraft projects. This usually consists of attitude orbit determinations, telemetry decommutation and other control functions. The software that is produced generally takes from six months to two years to produce, is written by two to ten programmers most of whom are working on several such projects simultaneously, and requires from six man-months to ten man-years of effort. Projects are supervised by NASA/GSFC employees and personnel are either NASA personnel or outside contractors (Computer Sciences Corporation).

The development facility consists of two primary hardware systems: a pair of S/360's and a PDP-11/70. During development of software systems users can expect turn-around time to vary from one or two hours for small, half-minute jobs, to one day for medium jobs (3 to 5 minutes, less that 600K), to several days for longer and larger jobs. The primary language used is FORTRAN although there is some application of assembler language.

THE META-MODEL

The meta-model described here is of the adjusted base-line type such as those proposed by Walston and Felix and Barry Boehm. Therefore, the basic approach is a two-step process. First, the effort expended for the average project is expressed as a function of some measure of size and, second, each project's deviation from this average is explained through the systematic use of a set of environmental attributes known for each project. The remainder of this paper will describe this process and will follow the format:

1) Compute the background equation
2) Analyze the factors available to explain the difference between actual effort and effort as predicted by the background equation
3) Use this model to predict the effort for the new project

The Background Equation

The background or base-line relationship between effort and size forms the basis for the local model. It is found by fitting some choice of curve through the scatter plot of effort versus size data. By definition, then, it should be able to predict the effort required to complete an average project, given its size. This average effort value as a function of size alone has been termed the "standard effort" throughout this paper. This section deals with:

1.1) Picking and defining measures of size and effort
1.2) Selecting the form of the base-line equation
1.3) Calculating an initial base-line for use in the model

In any given environment the decision of what size measure to use would have to depend initially upon what data is available. In our case, it was

we used can be applied. The study now involves complete data on eighteen projects and sub-projects but was begun when we had complete data on only five projects. It is hoped that the presentation of our work will save other investigators who are developing a model some time or at least provide a point of departure for their own study.

Background

There exist many cost estimation models ranging from highly theoretical ones, such as Putnam's model [1], to empirical ones, such as the Walston and Felix [2] and the Boehm model [3]. An empirical model uses data from previous projects to evaluate the current project and derives the basic formulae from analysis of the particular data base available. A theoretical model, on the other hand, uses formulae based upon global assumptions, such as the rate at which people solve problems, the number of problems available for solution at a given point in time, etc. The work in this paper is empirical and is based predominantly on the work of Walston and Felix, and Barry Boehm.

The Software Engineering Laboratory (SEL) has worked to validate some of the basic relationships proposed by Walston and Felix which dealt with the factors that affect the software development process. One result of their study was an index computed with twenty-nine factors they judged to have a significant effect on their software development environment. As part of their study, they proposed an effort equation which was of the form $E = 5.2 * L^{.91}$ where E is the total effort in man-months and L is the size in thousands of lines of delivered source code. Data from SEL was used to show that although the exact equation proposed by Walston and Felix could not be derived, the basic relationship between lines of code and effort could be substantiated by an equation which lay within one standard error of estimate for the IBM equation, and in a justifiable direction [10]. Barry Boehm has proposed a model that uses a similar standard effort equation and adjusts the initial estimates by a set of sixteen multipliers which are selected according to values assigned to their corresponding attributes. In attempting to fit an early version of this model, but with the SEL data, it was found that because of differing environments, a different baseline equation was needed, as well as a different set of environmental parameters or attributes. Many of the attributes found in the TRW environment are already accounted for in the SEL baseline equations, and several of the attributes in the SEL model which accounted for changes in productivity were not accounted for in the Boehm model, presumably because they had little effect in the TRW environment. Based upon this assumption and our experience with the IBM and TRW models, the meta model proposed in this paper was devised.

The SEL Environment

The Software Engineering Laboratory was organized in August, 1976. Beginning in November, 1976, most new software tasks that were assigned by the System Development Section of NASA/Goddard Space Flight Center began submitting data on development progress to our data base. These programs are mostly ground support routines for various spacecraft projects. This usually consists of attitude orbit determinations, telemetry decommutation and other control functions. The software that is produced generally takes from six months to two years to produce, is written by two to ten programmers most of whom are working on several such projects simultaneously, and requires from six man-months to ten man-years of effort. Projects are supervised by NASA/GSFC employees and personnel are either NASA personnel or outside contractors (Computer Sciences Corporation).

The development facility consists of two primary hardware systems: a pair of S/360's and a PDP-11/70. During development of software systems users can expect turn-around time to vary from one or two hours for small, half-minute jobs, to one day for medium jobs (3 to 5 minutes, less that 600K), to several days for longer and larger jobs. The primary language used is FORTRAN although there is some application of assembler language.

THE META-MODEL

The meta-model described here is of the adjusted base-line type such as those proposed by Walston and Felix and Barry Boehm. Therefore, the basic approach is a two-step process. First, the effort expended for the average project is expressed as a function of some measure of size and, second, each project's deviation from this average is explained through the systematic use of a set of environmental attributes known for each project. The remainder of this paper will describe this process and will follow the format:

1) Compute the background equation
2) Analyze the factors available to explain the difference between actual effort and effort as predicted by the background equation
3) Use this model to predict the effort for the new project

The Background Equation

The background or base-line relationship between effort and size forms the basis for the local model. It is found by fitting some choice of curve through the scatter plot of effort versus size data. By definition, then, it should be able to predict the effort required to complete an average project, given its size. This average effort value as a function of size alone has been termed the "standard effort" throughout this paper. This section deals with:

1.1) Picking and defining measures of size and effort
1.2) Selecting the form of the base-line equation
1.3) Calculating an initial base-line for use in the model

In any given environment the decision of what size measure to use would have to depend initially upon what data is available. In our case, it was

decided that size could be measured easily by lines of source code or by modules and that effort could be expressed in man-months. Consideration should also be given to the ease with which each measure can be estimated when the model is used to predict the effort required for future projects. The upper management in our programming environment was of the opinion that source lines with comments was the easier of the two readily available measures to predict. Also, it was decided that, based upon the data available and the ultimate use of the model, project effort would be defined to be measured from the beginning of the design phase through acceptance testing and to include programming, management and support hours.

In our data base, the total number of lines and modules as well as the number of new lines and new modules were available for the 18 projects and sub-projects. Initially, we expressed effort in terms of each of the four size measures mentioned above. To do this, we used three forms of equations to fit the data, using both the raw data and logarithms of the data, which provided functions we hoped would express the basic relationship between size and effort that exists in our environment. The forms of the three types of equations were:

E = effort S = size

$$E = a * S + b \quad (1)$$

$$E = a * S^b \quad (2)$$

$$E = a * S^b + c \quad (3)$$

Some difficulties were encountered when attempting to fit a conventional least-squares regression line through the raw data. One probable reason for this is that a correlation between the deviations from the prediction line and the size of the project could not easily be eliminated (heteroscedasticity). Rather than using a least-squares line with a single, arithmetic standard error of estimate which would be consistently large with respect to small projects and often too small when applying the equation to large projects, we opted for a prediction line which minimized the ratio between the predicted values for effort and each actual data point. In this way, the standard error is multiplicative and can be thought of as a percent error whose absolute magnitude increases as the project size increases. If, however, equations of the second or third form are derived by fitting a least-squares line through the logarithms of the data, the standard error automatically becomes multiplicative when converted back to linear coordinates.

The third form shown above was the most successful for us. It was in the form of an exponential fit but included a constant which removed the constraint that the prediction line pass through the origin. This line was not found by converting to logarithms but by an algorithm that selected the values which minimized the standard error of estimate when expressed as a ratio. The theory behind the implementation of this multiplicative standard error of estimate is described later. Although the size of our data base was not large enough to firmly support using this fit rather than a straight line, we are using it here primarily as an illustration, and therefore felt justified in retaining it.

Turning back to the measurement of size, it was noted that neither the equations based upon size in terms of new lines of code or new modules nor those based upon total lines of code or total modules captured the intuitive sense of the amount of work required for each project. It was felt that although using previously-written code was easier than generating new code, the integration effort was still significant and should be accounted for. After examining the background relationships discussed above, another more satisfying measurement for size was derived. Instead of considering only the total lines or only the new lines to determine the size of a project, an algorithm to combine these sizes into one measure was selected. It was found that by computing the effective size in lines to be equal to the total number of new lines written plus 20% of any old lines used in the project, a base-line relationship of lower standard error could be derived. This new size measure will be called "developed lines" in this paper. The same technique was applied to numbers of modules and resulted in a measure of "developed modules." Other proportions of new and old sizes were tried as well as an algorithm which computed developed size based on a graduated mixture of new and old code where larger projects counted a higher percentage of their re-used code in the developed size. Often, these equations did produce slightly better background relationships, but the improvement in standard error was judged not to be worth the added complexity. It was hoped that as long as some reasonable algorithm was selected which captured the size as measured by both the amount of new product as well as old product, most of the remaining differences among the projects should be explainable by the varying environmental attributes.

At this point, the three base-line equations, based on the computed sizes of developed lines only, were:

E = effort in man-months of programming and management time
DL = number of developed lines of source code with comments (new lines with comments plus 20% of re-used lines)

Equation:	*Standard error of estimate:	
$E = 1.36*DL + 1.62$	1.269	(4)
$E = 1.86*DL^{.93}$	1.297	(5)
$E = 0.73*DL^{1.16} + 3.5$	1.250	(6)

* Note that these are multiplicative factors. The predicted value given by the equation is multiplied and divided by this factor to get the range for one standard error of estimate. All standard errors of estimate (s.e.e.) in this paper are of this type.

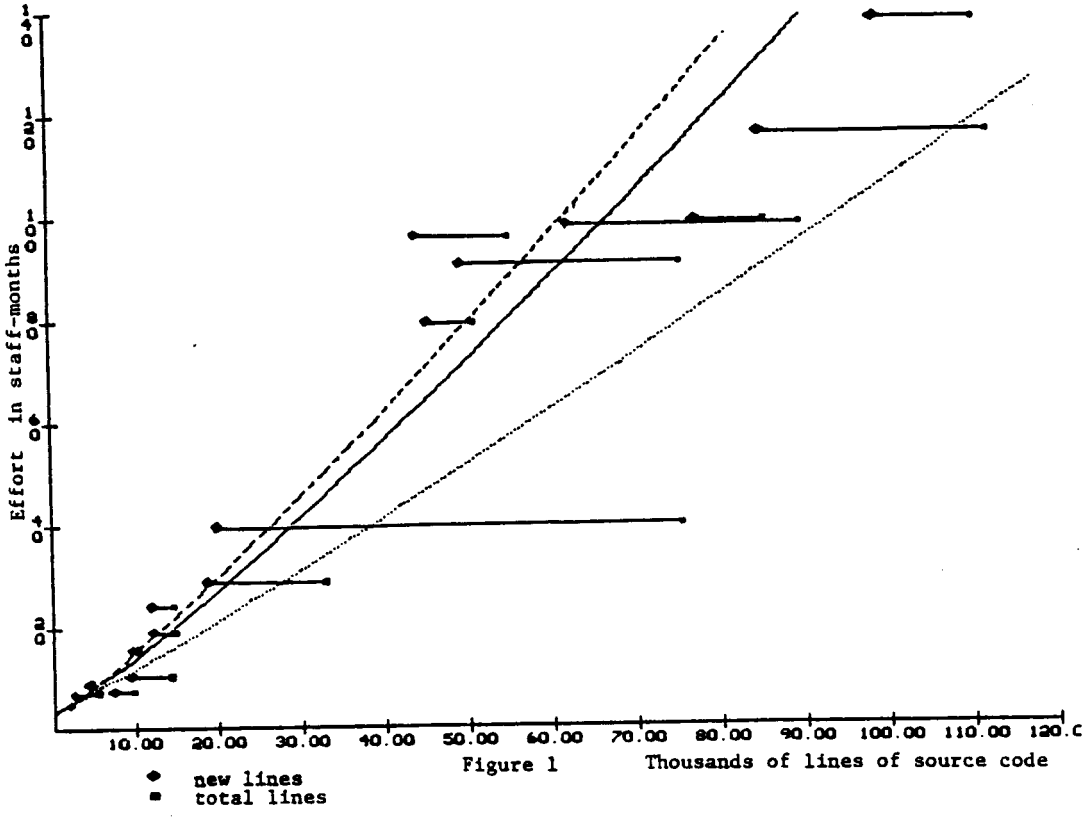


Figure 1

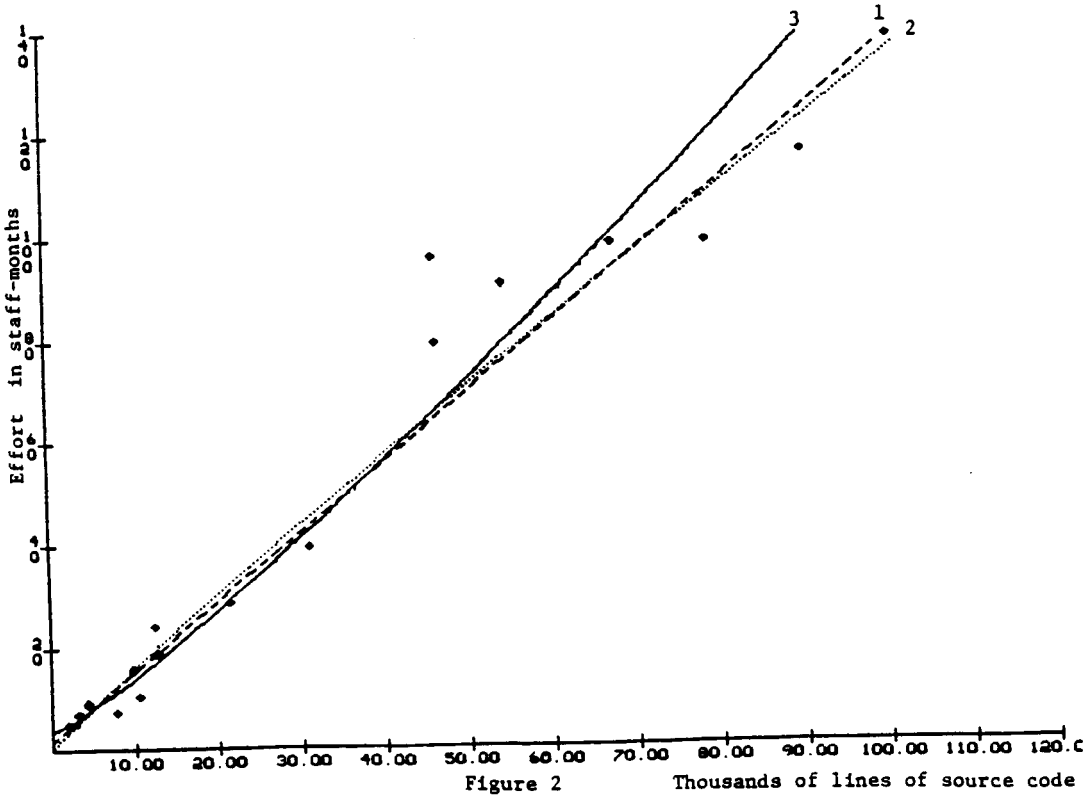


Figure 2

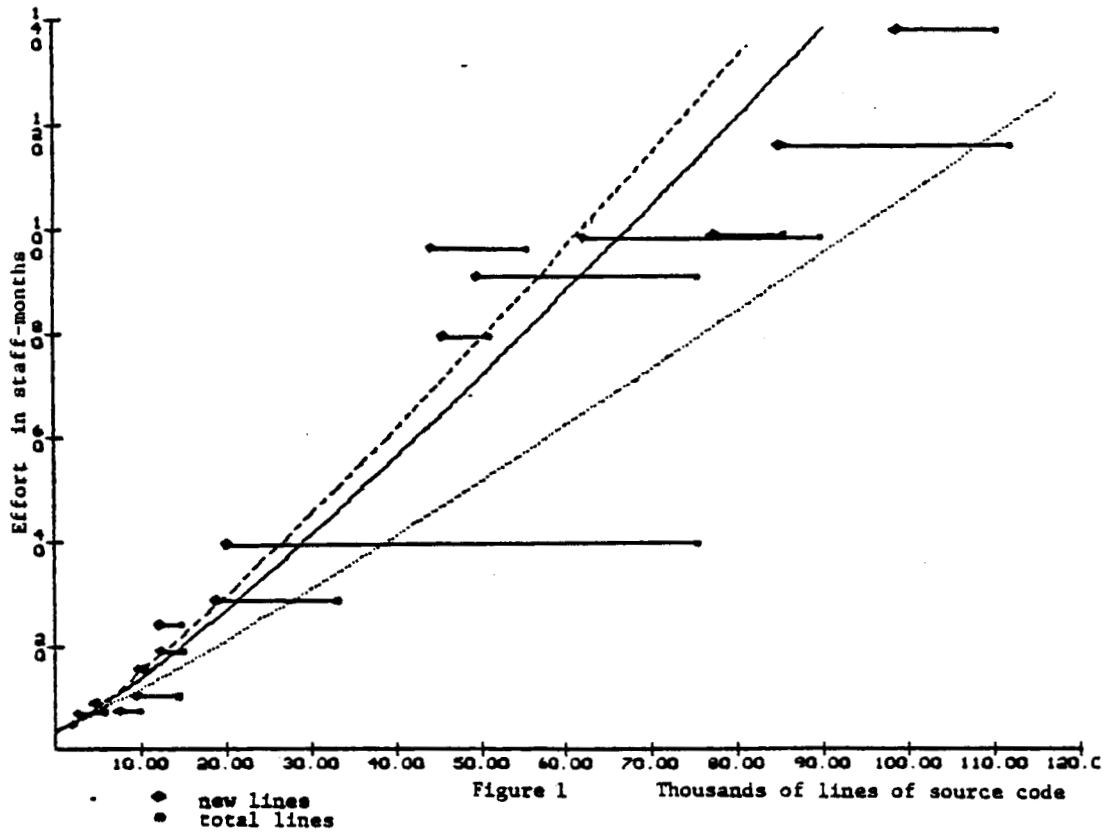


Figure 1

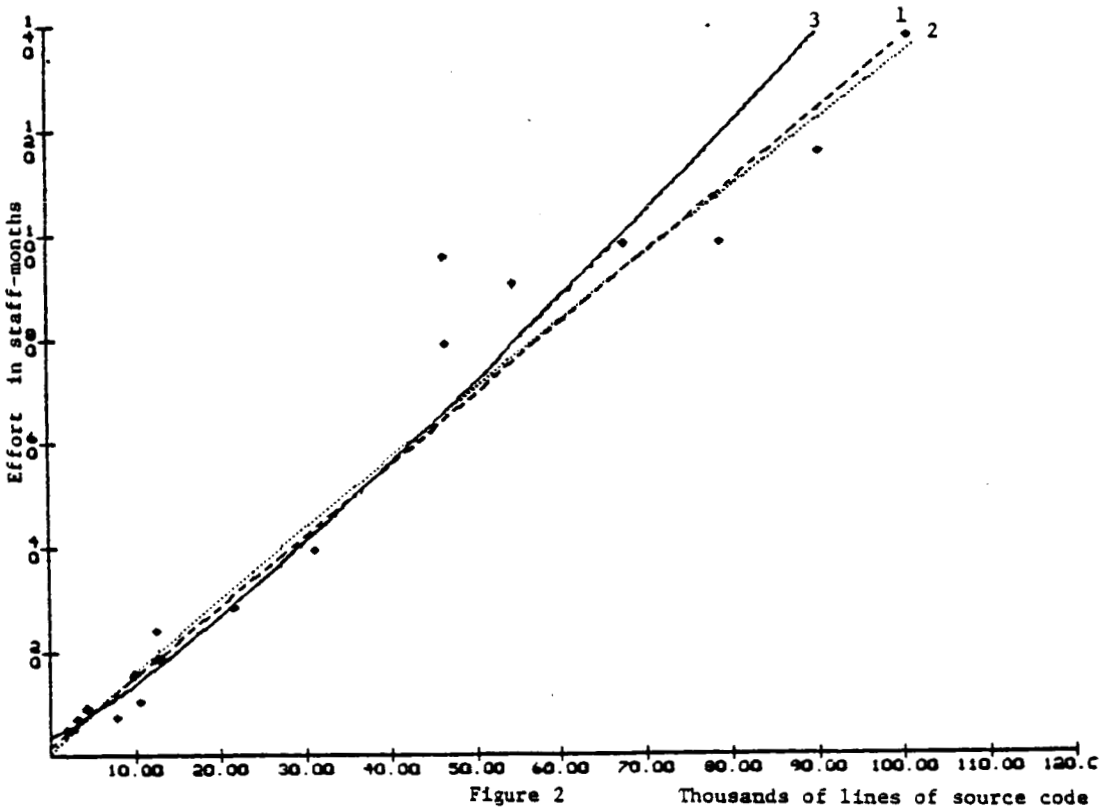


Figure 2

Figure 1 shows how the exponential fit with constant for developed lines falls between those for new lines and total lines, hopefully doing a better job than either of the other two in relating a project's size to the resources consumed during its development. The remainder of this paper will deal entirely with this computed measure of size since it was our most successful expression for work output for a given project.

Figure 2 shows these three background prediction equations superimposed on the data points. It was decided to use equation 3, above, as the base-line throughout the remainder of the model generation since it achieved the best fit to the data points and suggested the intuitively satisfying fact that a project requires a minimum overhead effort (the Y-intercept of the function). Equation one, a straight line, does as well statistically, and could well have been adopted for simplicity. Since this is meant to be an illustration, however, and it was felt that the non-linear relationship between size and effort was more common outside of our environment, equation three was adopted for use in this study. The remaining errors of estimation appear as the vertical distances between each point and the line. It is these distances in the form of ratios which we would like to explain in terms of the environmental attributes.

Project Factors

The next step in determining a model is to collect data about the programming environment of each project which captures the probable reasons why some projects took more effort and thereby consumed more resources than others when normalized for size. This data could include such factors as methodologies used during design and development, experience of the customer and of the programmers, managerial control during development, number of changes imposed during the development and type and complexity of the project. It is assumed that the correct application of information such as this can assist in explaining the variations observed among projects in terms of their productivities. The steps described in this section include:

2.1) Choosing a set of factors
2.2) Grouping and compressing this data
2.3) Isolating the important factors and groups
2.4) Incorporating the factors by performing a multiple regression to predict the deviations of the points from the computed base-line

In all, close to one hundred environmental attributes were examined as possible contributors to the variations among the productivities of the projects. Table 1 shows a list of these factors as well as some others which we did not use. Thirty-six of the factors were those used by Walston and Felix, sixteen were used by Boehm and 30 others were suggested by our environment. Although we did not use all these factors, they are included to provide additional ideas for other investigators. It should be noted that it is not necessary to consider any factors which are constant for the set of projects currently in the data-base since the influence of this factor will already be contained in the base-line relationship. If, however, a future project is rated differently in one of these categories, it may be necessary to reinstate it into the model.

The process of selecting attributes to use is largely a matter of what information is available. Since many of the projects we studied were completed when this investigation began, it was necessary to rely on project management for the information required. The inclusion of past projects was justified in order to establish as large a data-base as possible, however, it made it necessary to be particularly careful about the consistency between the ratings for current projects and those for projects already completed. To maintain the integrity of the values of these attributes, all ratings produced by the vendor's management were examined by the customer's management and also by us. In this way we hoped to avoid the temptation to adjust ratings to reflect the known ultimate success of past projects.

Many of the attributes required no special work to assign a value, such as "Team Size" or "Percent Code: I/O," but most required imposing a scale of some kind. We decided that an exact scale was not possible or even necessary so a six-point subjective rating was used. This format was chosen by the managers who would be making the ratings since it conformed well with the information they had already collected about many of the attributes. Most of the factors, then, are rated on a scale from 0 to 5 with 5 being the most of that particular attribute (whether it is "good" or "bad"). The most important point is that we tried to remain consistent in our ratings from project to project. The need for this was particularly noticeable when rating earlier projects in terms of development methodology. For instance, what may have been thought of as a "4" rating in "Formal Training" for a project which began coding over a year ago may actually be a "3" or even a "2" when compared with the increased sophistication of more recent projects. We found it necessary to re-scale a few of the attributes because of this consideration.

After a set of environmental factors is selected and the data collected, it is necessary to consider the number of these attributes versus the number of projects in the data base. It is not statistically sound to use a large group of factors to predict a variable with relatively few data points. Unless a very large number of projects is being used, it will probably be necessary to condense the information contained in the whole set of factors into just a few new factors. This can be accomplished entirely intuitively, based on experience, or with the help of a correlation matrix or factor analysis routines. Although there is no absolute rule as to how many factors should be used to predict a given number of points, a rule of thumb might be to allow up to ten or fifteen percent of the number of data points. Strictly speaking, the adjusted r-squared values or the F-values should be observed as factors are added to the prediction equation via a multiple regression routine (described below) to avoid the mistake of using too many factors.

In our environment, we had data on 71 attributes which we suspected could affect the ultimate

Walston and Felix:	Boehm:
	Required fault freedom
Customer experience	Data base size
Customer participation in definition	Product complexity
Customer interface complexity	Adaptation from existing software
Development location	Execution time constraint
Percent programmers in design	Main storage constraint
Programmer qualifications	Virtual machine volatility
Programmer experience with machine	Computer response time
Programmer experience with language	Analyst capability
Programmer experience with application	Applications experience
Worked together on same type of problem	Programmer Capability
Customer originated program design changes	Virtual machine experience
Hardware under development	Programming language experience
Development environment closed	Modern programming practices
Development environment open with request	Use of software tools
Development environment open	Required development Schedule
Development environment RJE	
Development environment TSO	SEL:
Percent code structured	Program design language (development and design)
Percent code used code review	Formal design review
Percent code used top-down	Tree charts
Percent code by chief-programmer teams	Design formalisms
Complexity of application processing	Design/decision notes
Complexity of p.ogram flow	Walk-through: design
Complexity of internal communication	Walk-through: code
Complexity of external communication	Code reading
Complexity of data-base structure	Top-down design
Percent code non-math and I/O	Top-down code
Percent code math and computational	Structured code
Percent code CPU and I/O control	Librarian
Percent code fallback and recovery	Chief Programmer Teams
Percent code other	Formal Training
Proportion code real time of interactive	Formal test plans
Design constraints: main storage	Unit development folders
Design constraints: timing	Formal documentation
Design constraints: I/O capability	Heavy management involvement and control
Unclassified	Iterative enhancement
	Individual decisions
	Timely specs and no changes
	Team size
	On schedule
	TSO development
	Overall
	Reusable code
	Percent programmer effort
	Percent management effort
	Amount documentation
	Staff size

Table 1

productivity of a project, but only 18 projects for which to see the results. We found it necessary, therefore, to perform such a compression of the data. Our next step, then, was to examine the attributes and group into categories those which we felt would have a similar effect on the project. As an aid to selecting potential groupings for analysis, a correlation matrix for all the attributes was studied. It was hoped that meaningful groups could be formed which would retain an intuitive sense of positive or negative contribution to the project's productivity. By studying the potential categorizations of the factors, and how they performed in potential models to predict developed lines, we settled upon three groups using 21 of the original attributes. The groups and their constituent attributes were:

Total Methodology
- Tree Charts
- Top Down Design
- Design Formalisms
- Formal Documentation
- Code Reading
- Chief Programmer Teams
- Formal Test Plans
- Unit Development Folders
- Formal Training

Walston and Felix:	Boehm:
	Required fault freedom
Customer experience	Data base size
Customer participation in definition	Product complexity
Customer interface complexity	Adaptation from existing software
Development location	Execution time constraint
Percent programmers in design	Main storage constraint
Programmer qualifications	Virtual machine volatility
Programmer experience with machine	Computer response time
Programmer experience with language	Analyst capability
Programmer experience with application	Applications experience
Worked together on same type of problem	Programmer Capability
Customer originated program design changes	Virtual machine experience
Hardware under development	Programming language experience
Development environment closed	Modern programming practices
Development environment open with request	Use of software tools
Development environment open	Required development Schedule
Development environment RJE	
Development environment TSO	SEL:
Percent code structured	Program design language (development and design)
Percent code used code review	Formal design review
Percent code used top-down	Tree charts
Percent code by chief-programmer teams	Design formalisms
Complexity of application processing	Design/decision notes
Complexity of program flow	Walk-through: design
Complexity of internal communication	Walk-through: code
Complexity of external communication	Code reading
Complexity of data-base structure	Top-down design
Percent code non-math and I/O	Top-down code
Percent code math and computational	Structured code
Percent code CPU and I/O control	Librarian
Percent code fallback and recovery	Chief Programmer Teams
Percent code other	Formal Training
Proportion code real time of interactive	Formal test plans
Design constraints: main storage	Unit development folders
Design constraints: timing	Formal documentation
Design constraints: I/O capability	Heavy management involvement and control
Unclassified	Iterative enhancement
	Individual decisions
	Timely specs and no changes
	Team size
	On schedule
	TSO development
	Overall
	Reusable code
	Percent programmer effort
	Percent management effort
	Amount documentation
	Staff size

Table 1

productivity of a project, but only 18 projects for which to see the results. We found it necessary, therefore, to perform such a compression of the data. Our next step, then, was to examine the attributes and group into categories those which we felt would have a similar effect on the project. As an aid to selecting potential groupings for analysis, a correlation matrix for all the attributes was studied. It was hoped that meaningful groups could be formed which would retain an intuitive sense of positive or negative contribution to the project's productivity. By studying the potential categorizations of the factors, and how they performed in potential models to predict developed lines, we settled upon three groups using 21 of the original attributes. The groups and their constituent attributes were:

Total Methodology
- Tree Charts
- Top Down Design
- Design Formalisms
- Formal Documentation
- Code Reading
- Chief Programmer Teams
- Formal Test Plans
- Unit Development Folders
- Formal Training

Cumulative Complexity
- Customer Interface Complexity
- Customer-Initiated Design Changes
- Application Process Complexity
- Program Flow Complexity
- Internal Communication Complexity
- External Communication Complexity
- Data Base Complexity

Cumulative Experience
- Programmer Qualifications
- Programmer Experience with Machine
- Programmer Experience with Language
- Programmer Experience with Application
- Team Previously Worked Together

We were particularly interested in using a methodology category due to the findings of Basili and Reiter [11] which implied improvement in the development process due to the use of a specific discipline. The methodology category was selected to closely coincide with the principles of the methodology used in the experiment. The complexity category was included to account for some of the known negative influences on productivity. The cumulative rating for each of these categories was merely a sum of the ratings of its constituents (each adjusted to a 0 to 5 scale). Although it was necessary to reduce the number of attributes used in the statistical investigation in this manner in order to give more meaningful results, the simple summing of various attributes loses some of the information which could be reflected in these categories. This is because even though one of the constituent attributes may be much more important than another, an unweighted sum will destroy this difference. One solution to this type of dilemma is to have many more data points, as mentioned before, and to use the attributes independently. Another would be to determine the relative effects of each attribute and to weight them accordingly. Without the necessary criteria for either of these solutions, however, we were forced to continue in this direction and to accept this trade-off.

Incorporating the Factors

The purpose of the attribute analysis is to explain the deviations displayed by each project from the derived background equation and, ultimately, to yield a prediction process where the attributes can be used to determine how far a project will "miss" the background equation, if at all.

The next step, then, is to compute these differences which must be predicted. A quantity based on the ratio between the actual effort expended and the amount predicted by the background equation was used as a target for the prediction. In this way, when the model is in use, the background equation can be applied to determine the standard effort (the amount needed if the project behaved as an average of the previous projects in the data-base). Then, the attributes will be used to yield a ratio between this rough estimate and a hopefully more accurate expected value of the effort required.

The SPSS [12] forward multiple regression routine was used to generate an equation which could best predict each of the project's ratio of error. The actual ratio was converted to a linear scale with zero meaning the actual data point fell on the base line. This was accomplished by subtracting one from all ratios greater than one and adding one to the negative reciprocals of those ratios which were less than one. For instance, if a project's standard effort was predicted to be 100 man-months and it actually required 150 man-months, this ratio would be 1.5. Subtracting one makes this project's target value 0.5. If however it had needed only 66.7 man-months, its ratio would be .667 which is less than one. Adding one to the negative reciprocal of this number gives a target value of -0.5. The assumption is that this scale tends to be symmetrical in that the first project had as many negative factors impact its productivity as the second project had positive.

In the first pass at using the multiple regression routine, we were using five attribute groups. Since the data base was not very large, we were cautious about assigning any useful significance to the results. We therefore recondensed the attribute data into the three groups shown above. The results of this attempt are described in a later section.

Variations on the Model

We noticed that it was possible to combine the two processes of first isolating a background equation and then applying the environmental attributes to explain deviations from that equation into a single procedure. To do this, a measure of size was included as a factor with the set of environmental attributes and the whole group was used to predict effort. As expected, size was always chosen first by the forward routine, since it correlated the best with effort for each project. This single process lacked the intuitively satisfying intermediate stage which related to a base-line relationship as a half-way point in the model's results, but it streamlined the model somewhat.

In order to preserve the possibility of an exponential relationship between size and effort, this method was used with the logarithms of the size and effort values. The output of the regression analysis would be of the form,

$$\log(\text{Effort}) = A*\log(\text{Size}) + B*\text{attr1} + C*\text{attr2} + \ldots + K \quad (7)$$

This would convert to,

$$\text{Effort} = \text{Size}^{A} * 10^{(B*\text{attr1}+C*\text{attr2}+\ldots+K)} \quad (8)$$

assuming, here, that log base 10 was used in the conversion.

A third template for a model was tried which attempted to eliminate nearly all of the reliance on the actual numerical values of our attribute ratings in order to legitimize some of our statistical analyses. Only two of the attribute groups mentioned before were considered, "Complexity" and "Methodology." Each of these two ratings were transformed into two new ratings of binary values resulting in four new attributes, "High Methodol-

ogy," "Low Methodology," "High Complexity," and "Low Complexity." The transformation was accomplished as follows: if a project's rating fell in the upper third of all projects, the value of the "High" binary attribute of that type was assigned a 1 while the value of the "Low" attribute for that type was assigned a 0. If the value fell in the middle third, both binary values were assigned a 0. If the value fell in the low third, the "Low" attribute was assigned a 1 while the "High" was assigned a 0. This reduced our assumptions about the data to the lowest level for statistical analysis. For illustration, call the four new binary attributes HM, LM, HC, LC for high and low methodology and high and low complexity. The result of the multiple regression analysis, then, would be in the form,

$$\text{Effort} = \text{Size}^{A} * 10^{(B*HM+C*LM+D*HC+E*LC+K)} \quad (9)$$

Since the chance that any chosen attribute value will be 0 for a particular project is about 2/3, most of those terms on the right will drop out when the model is actually applied to a given project. Although we did not expect to achieve the same accuracy from this method, the simplicity of it was appealing.

APPLYING THE MODEL

As an illustration of the results obtained thus far for our environment, this section deals with the actual values of the data we used and the models we generated. It should serve as a useful guide and a summary of the steps we chose to follow. In order to include an illustration of the functioning of the completed model, one project, the most recently completed project, will be removed from the analysis while a new model is developed. This project will then be treated as a new data point in order to test and illustrate the performance of the model. Typically, the use of the model will involve the following steps:

3.1) Estimate size of new project
3.2) Use base-line to get standard effort
3.3) Estimate necessary factor values
3.4) Compute difference this project should exhibit
3.5) Apply that difference to standard effort

Appendix 1 shows the eighteen projects and sub-projects currently in our data-base with the measures of size previously discussed. As stated above, developed size is all of the newly-written lines or modules plus 20% of the re-used lines or modules, depending on which size measure is being used. The developed size is what we chose to predict with the models generated. We also chose, as a baseline, the exponential equation with the constant term. The following illustration shows the development of the model with the first seventeen points in the data base. The base-line relationship between developed lines of code and effort was:

$$E = .72 * DL^{1.17} + 3.4 \quad (s.e.e.=1.25) \quad (10)$$

The remaining information used about the projects is shown in the appendix. The remaining error ratios from this line to each project's actual effort were computed and listed. These are the values which should be explained by the multiple regression analysis. When the model is in use, then, an error ratio can be derived by using the multiple regression equation which can then be applied to the base-line equation to provide what should be an even better estimate of effort than the base-line alone. As discussed, the three main categories of environmental attributes shown are the result of distilling many attributes.

The equations computed by the SPSS forward multiple regression routine which attempt to express the list of error ratios as functions of various of the attributes provided are:

ER = Effort ratio (converted to linear scale)
METH = Methodology
CMPLX = Complexity

$$ER = -.036 * METH + 1.0 \quad (11)$$

$$ER = -.036 * METH + .006 * CMPLX + .86 \quad (12)$$

To apply the model to the unused, eighteenth point, the base-line equation is first used to establish the standard effort. Since the estimated size of the project was 101,000 lines, this standard effort value was 163 man-months with a range for one standard error of from 130 to 204 man-months. When the additional attributes are used to compute the error ratio as given by the multiple regression equations, the results (for each of the above equations) are:

$$ER = -0.224$$

$$ER = -0.166$$

Converting these numbers back to multiplicative factors means dividing the standard effort by 1.224 and by 1.166, respectively. When these ratios are applied to the standard effort value, the revised effort values are found to be 133 man-months with a range for one standard error from 115 to 154 man-months for the first equation, and 140 man-months with a range for one standard error of from 121 to 162 man-months for the second equation. The actual effort for the project is known to have been 138 man-months.

Once any new project is added to the data base, at least the generation of the base-line relationship and the multiple regression analysis of the error ratios should be repeated. It may also be necessary to examine the factor groupings to see if they could be modified to increase the accuracy of the model or to include a previously unimportant attribute.

For our data, when this eighteenth point is added to the data base, the base-line equation becomes:

$$E = .73 * DL^{1.16} + 3.5 \quad (s.e.e.=1.25) \quad (13)$$

while the equations to predict the error ratio from the attributes become:

ogy," "Low Methodology," "High Complexity," and "Low Complexity." The transformation was accomplished as follows: if a project's rating fell in the upper third of all projects, the value of the "High" binary attribute of that type was assigned a 1 while the value of the "Low" attribute for that type was assigned a 0. If the value fell in the middle third, both binary values were assigned a 0. If the value fell in the low third, the "Low" attribute was assigned a 1 while the "High" was assigned a 0. This reduced our assumptions about the data to the lowest level for statistical analysis. For illustration, call the four new binary attributes HM, LM, HC, LC for high and low methodology and high and low complexity. The result of the multiple regression analysis, then, would be in the form,

$$\text{Effort} = \text{Size}^{A} * 10^{(B*HM+C*LM+D*HC+E*LC+K)} \quad (9)$$

Since the chance that any chosen attribute value will be 0 for a particular project is about 2/3, most of those terms on the right will drop out when the model is actually applied to a given project. Although we did not expect to achieve the same accuracy from this method, the simplicity of it was appealing.

APPLYING THE MODEL

As an illustration of the results obtained thus far for our environment, this section deals with the actual values of the data we used and the models we generated. It should serve as a useful guide and a summary of the steps we chose to follow. In order to include an illustration of the functioning of the completed model, one project, the most recently completed project, will be removed from the analysis while a new model is developed. This project will then be treated as a new data point in order to test and illustrate the performance of the model. Typically, the use of the model will involve the following steps:

3.1) Estimate size of new project
3.2) Use base-line to get standard effort
3.3) Estimate necessary factor values
3.4) Compute difference this project should exhibit
3.5) Apply that difference to standard effort

Appendix 1 shows the eighteen projects and sub-projects currently in our data-base with the measures of size previously discussed. As stated above, developed size is all of the newly-written lines or modules plus 20% of the re-used lines or modules, depending on which size measure is being used. The developed size is what we chose to predict with the models generated. We also chose, as a baseline, the exponential equation with the constant term. The following illustration shows the development of the model with the first seventeen points in the data base. The base-line relationship between developed lines of code and effort was:

$$E = .72 * DL^{1.17} + 3.4 \quad (s.e.e.=1.25) \quad (10)$$

The remaining information used about the projects is shown in the appendix. The remaining error ratios from this line to each project's actual effort were computed and listed. These are the values which should be explained by the multiple regression analysis. When the model is in use, then, an error ratio can be derived by using the multiple regression equation which can then be applied to the base-line equation to provide what should be an even better estimate of effort than the base-line alone. As discussed, the three main categories of environmental attributes shown are the result of distilling many attributes.

The equations computed by the SPSS forward multiple regression routine which attempt to express the list of error ratios as functions of various of the attributes provided are:

ER = Effort ratio (converted to linear scale)
METH = Methodology
CMPLX = Complexity

$$ER = -.036 * METH + 1.0 \quad (11)$$

$$ER = -.036 * METH + .006 * CMPLX + .86 \quad (12)$$

To apply the model to the unused, eighteenth point, the base-line equation is first used to establish the standard effort. Since the estimated size of the project was 101,000 lines, this standard effort value was 163 man-months with a range for one standard error of from 130 to 204 man-months. When the additional attributes are used to compute the error ratio as given by the multiple regression equations, the results (for each of the above equations) are:

$$ER = -0.224$$

$$ER = -0.166$$

Converting these numbers back to multiplicative factors means dividing the standard effort by 1.224 and by 1.166, respectively. When these ratios are applied to the standard effort value, the revised effort values are found to be 133 man-months with a range for one standard error from 115 to 154 man-months for the first equation, and 140 man-months with a range for one standard error of from 121 to 162 man-months for the second equation. The actual effort for the project is known to have been 138 man-months.

Once any new project is added to the data base, at least the generation of the base-line relationship and the multiple regression analysis of the error ratios should be repeated. It may also be necessary to examine the factor groupings to see if they could be modified to increase the accuracy of the model or to include a previously unimportant attribute.

For our data, when this eighteenth point is added to the data base, the base-line equation becomes:

$$E = .73 * DL^{1.16} + 3.5 \quad (s.e.e.=1.25) \quad (13)$$

while the equations to predict the error ratio from the attributes become:

$$ER = -.035 * METH + .98 \quad (s.e.e.=1.16) \quad (14)$$

$$ER = -.036 * METH + .009 * CMPLX + .80 \quad (s.e.e.=1.15) \quad (15)$$

It should be remembered that the original choice of factors from the entire set, and the groupings of these factors, was done with regard to predicting size as measured by developed lines and was not so specifically tuned to predicting developed modules. It is reasonable to expect, then, that the results of the models generated to predict effort from the number of developed modules using these attribute groupings will be less accurate than those using the number of developed lines. If the objective had been to generate a model specifically suited to predicting modules, various adjustments would have been made during the early part of the model's development. Also, it is advisable to review the model each time a new project is completed and its data is added to the data base. In this way the model can be refined and kept up-to-date, and will be able to take into account changes in the overall programming environment.

Although we are not reporting here the actual values and equations generated in the development of the other forms of this basic model (described under "Variations on the Model," above) it became apparent that none of the model types is by far better than the rest, especially considering the fact that they all have differing amounts of statistical significance. In terms of a purely investigative study, all of them should probably be examined further. As more environmental information is added to the data-base, it may be possible to reorganize the constituent groups involved in the environmental attributes and to produce better categories. Also, when several more projects are completed, it may be possible to justifiably expand the size of the set of variables used to predict the expected value in the multiple regression routine giving the potential for greater accuracy.

CONCLUSIONS

There is reason to believe that the techniques outlined here and used in our laboratory have potential in terms of producing a useful model which is specifically developed for use at any particular environment. The main difficulty seems to be in determining which environmental attributes really capture the reason for the differences in productivity among the projects. The use of too few of these attributes will mean less of the variation can possibly be explained, while the use of too many makes the analysis statistically meaningless. We found that it was necessary to stop including factors with the multiple regression analysis when the r-squared value indicated that we had explained no more than half of the variations among the error ratios. This would seem to indicate that there were considerably more influences upon the productivities of the projects than we managed to isolate. Simplifying the original idea for the model, however, which reduced the emphasis on the quality of the data did not weaken the accuracy of the model beyond useful proportions. This is particularly important when so much of the data which is essential to build the model is subjective and consequently non-linear.

Acknowledgements: The authors would like to thank Dr. Jerry Page of Computer Sciences Corporation and Frank McGarry of NASA/Goddard Space Flight Center for their invaluable help in providing the data for this study.

Research for this study was supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

References

(1) Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering 4, No. 4, 1978.

(2) Walston, C. and Felix, C., "A Method of Programming Measurement and Estimation," IBM Systems Journal 16, Number 1, 1977.

(3) Boehm, Barry W., Draft of book on Software Engineering Economics, to be published.

(4) Lawrence, M. J. and Jeffery, D. R., "Inter-organizational Comparison of Programming Productivity," Department of Information Systems, University of New South Wales, March 1979.

(5) Doty Associates, Inc., Software Cost Estimates Study, Vol. 1, RADC TR 77-220, June 1977.

(6) Wolverton, R., "The Cost of Developing Large Scale Software," IEEE Transactions on Computers 23, No. 6, 1974

(7) Aron, J., "Estimating Resources for Large Programming Systems," NATO Conference on Software Engineering Techniques, Mason Charter, N. Y. 1969.

(8) Carriere, W. M. and Thibodeau, R., "Development of a Logistics Software Cost Estimating Technique for Foreigh Military Sales," General Research Corporation, Santa Barbara, California, June 1979.

(9) Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (Ed.) Penguin Books, Inc., Baltimore, Md. 1970, pp. 77-101.

(10) Basili, V. R. and Freburger, K, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, Vol. 2, No. 1, 1981.

(11) Basili, V. R. and Reiter, R. W. Jr., "An Investigation of Human Factors in Software Development," Computer Magazine, December 1979, pp. 21-38.

(12) Statistical Package for the Social Sciences, Univac 1100 series manual.

Appendix 1

Project	Effort (man-months)	Total Lines	New Lines	Developed Lines	Predicted Standard Effort	Effort Ratio Standard/ Actual	Method-ology	Complex-ity	Exper-ience
1	115.8	111.9	84.7	90.2	138.7	.835	30	21	16
2	96.0	55.2	44.0	46.2	65.8	1.459	20	21	14
3	79.0	50.9	45.3	46.5	66.2	1.194	19	21	16
4	90.8	75.4	49.3	54.5	79.0	1.150	20	29	16
5	39.6	75.4	20.1	31.1	42.9	.924	35	21	18
6	98.4	89.5	62.0	97.5	100.1	.982	29	29	14
7	18.9	14.9	12.2	12.8	17.5	1.082	26	25	16
8	10.3	14.3	9.6	10.5	14.7	.704	34	19	21
9	28.5	32.8	18.7	21.5	29.2	.977	31	27	20
10	7.0	5.5	2.5	3.1	6.2	1.128	26	18	6
11	9.0	4.5	4.2	4.2	7.4	1.220	19	23	12
12	7.3	9.7	7.4	7.8	11.4	.640	31	18	16
13	5.0	2.1	2.1	2.1	5.2	.957	28	19	20
14	8.4	5.2	4.9	5.0	8.2	1.025	29	21	14
15	98.7	85.4	76.9	78.6	118.8	.831	35	33	16
16	15.6	10.2	9.6	9.7	13.7	1.138	27	21	16
17	23.9	14.8	11.9	12.5	17.1	1.398	27	23	18
18	138.3	110.3	98.4	100.8	157.4	.879	34	33	16

Appendix 1

Project	Effort (man-months)	Total Lines	New Lines	Developed Lines	Predicted Standard Effort	Effort Ratio Standard/ Actual	Method-ology	Complex-ity	Exper-ience
1	115.8	111.9	84.7	90.2	138.7	.835	30	21	16
2	96.0	55.2	44.0	46.2	65.8	1.459	20	21	14
3	79.0	50.9	45.3	46.5	66.2	1.194	19	21	16
4	90.8	75.4	49.3	54.5	79.0	1.150	20	29	16
5	39.6	75.4	20.1	31.1	42.9	.924	35	21	18
6	98.4	89.5	62.0	97.5	100.1	.982	29	29	14
7	18.9	14.9	12.2	12.8	17.5	1.082	26	25	16
8	10.3	14.3	9.6	10.5	14.7	.704	34	19	21
9	28.5	32.8	18.7	21.5	29.2	.977	31	27	20
10	7.0	5.5	2.5	3.1	6.2	1.128	26	18	6
11	9.0	4.5	4.2	4.2	7.4	1.220	19	23	12
12	7.3	9.7	7.4	7.8	11.4	.640	31	18	16
13	5.0	2.1	2.1	2.1	5.2	.957	28	19	20
14	8.4	5.2	4.9	5.0	8.2	1.025	29	21	14
15	98.7	85.4	76.9	78.6	118.8	.831	35	33	16
16	15.6	10.2	9.6	9.7	13.7	1.138	27	21	16
17	23.9	14.8	11.9	12.5	17.1	1.398	27	23	18
18	138.3	110.3	98.4	100.8	157.4	.879	34	33	16

Can the Parr Curve Help with Manpower Distribution and Resource Estimation Problems?*

Victor R. Basili and John Beane
Department of Computer Science, University of Maryland

This paper analyzes the resource utilization curve developed by Parr. The curve is compared with several other curves, including the Rayleigh curve, a parabola, and a trapezoid, with respect to how well they fit manpower utilization. The evaluation is performed for several projects developed in the Software Engineering Laboratory of the 6–12 man-year variety. The conclusion drawn is that the Parr curve can be made to fit the data better than the other curves. However, because of the noise in the data, it is difficult to confirm the shape of the manpower distribution from the data alone and therefore difficult to validate any particular model. Also, since the parameters used in the curve are not easily calculable or estimable from known data, the curve is not effective for resource estimation.

INTRODUCTION

Two important problems face the project manager at the beginning of the software development process. First, the manager must estimate the basic quantities of concern: the cost of the system, the duration of the project, and the size of the development team. The techniques for estimating cost have received more attention, but perhaps the crucial quantity in determining the success of the project is the schedule. The initial estimate of duration is often incapable of being changed because many contracts now include deadlines, with financial penalties for missing them. The mistake of underestimating the project duration can have dire effects. Brooks [1] points out that the common practice of increasing the production team when a project is late can involve more trouble than benefit. Putnam [2] has presented a model that illustrates in a quantitative way that the tradeoff of manpower for time is not free. Further, there are limits as to how far a schedule can be shortened depending on the difficulty of the development effort. Scheduling decisions cannot be made arbitrarily as a matter of convenience.

Once the estimates of the project cost, schedule, and team size are made, the next problem facing the project manager is how to distribute the total effort (represented by cost and team size) over the course of the project schedule. This problem has been solved for some large-scale projects using the Putnam model. Previous work has been done at the Software Engineering Laboratory (SEL) at the University of Maryland to decide whether the early prototype of the Putnam model, designed for large projects, could be applied to small- and medium-scale projects as well. The results have been mixed. To understand better why this model is less effective, it is important to consider the characteristics of the SEL environment.

The Software Engineering Laboratory collects and analyzes the data from projects built by Computer Sciences Corporation for the Goddard Space Flight Center (NASA). The goals of the laboratory are

1. to provide management with a mechanism to monitor the status of current projects;
2. to collect data to study the software process, to find what parameters can be isolated (understood), and to build measures incorporating these parameters; and
3. to compare the effects of various techniques upon system development [3, p. 116].

The seven projects used in this study are all attitude determination packages for satellite systems. They range in size from 50,900 to 111,900 lines of delivered source code (including comments). The code is mostly written in FORTRAN, with a small portion written in assembly language. The cumulative effort varied from 3

*Research supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

Address correspondence to Victor R. Basili, Department of Computer Science, University of Maryland, College Park, Maryland 20742.

The Journal of Systems and Software 2, 59–69 (1981)

Table 1. Statistics About the Projects

	Project						
	1	2	3	4	5	6	7
Total lines[a]	111.9	55.2	50.9	75.4	75.4	85.4	89.5
New lines[a]	84.7	44.0	45.3	49.3	20.1	76.9	62.0
Developed lines[a]	90.2	46.2	46.5	54.5	31.1	78.6	67.5
Effort (man-months)	115.7	95.9	78.9	90.7	39.6	98.6	98.3
Duration (months)	15.8	11.5	13.2	12.5	8.7	17.4	14.3
Average staff size	7.3	8.3	6.0	7.3	4.6	5.7	6.9

[a]In thousands of lines of source code.

to 10 man-years and lasted 9–18 months. A complete set of statistics is given in Table 1. All these projects fall into the medium-size range. It should be noted that new projects are often upgraded versions or other modifications of existing systems. The implications of this are twofold. Many projects can use some of the design and even the code of previous systems, and the organization as a whole has great experience with the application area (since for them the problems are well defined). In contrast, large-scale projects can be characterized as needing more than "2 years of development time, 50 man-years of effort or greater, and a hierarchical management structure of several layers" [2, p. 302].

The work described in this paper is a continuation of the studies of Basili and Zelkowitz [3] and Mapp [4]. Their analysis can be divided into two parts. In the first part, they asked whether the Putnam model could be used as an estimation tool. They took the Rayleigh equation (which is the central part of the Putnam model) and derived a relationship among three important quantities of the software process: the total effort K, in man-hours; the number of weeks T_s until acceptance testing; and the maximum staffing Y_d, in man-hours per week. During the requirements phase the contractor estimates each quantity, and the data are reported on the general project summary (GPS) form. Given any two of these estimates, a prediction of the third quantity can be based on the Rayleigh equation. The most interesting quantity (as we mentioned before) is the project duration, since NASA budgets fix the total resources each year in advance, and the contractor assigns a fixed number of people to the effort. The predictions of the time to acceptance testing were quite good when compared with the actual dates, in contrast to the original estimates [3]. The GPS estimates were consistently too low. Thus, the Rayleigh equation provides a check to ensure the project duration is not underestimated.

The second part of the analysis considered how well the Rayleigh curve fit the shape of actual staffing data over time. The Rayleigh equation can be rewritten in the form of a line for the variables y/t and t^2. After this, a line can be fit to the transformed data using linear regression. When Basili and Zelkowitz tried this, they found that the resulting curves did not follow the general shape of the data. At a glance it was clear that other curves could have fit the data better, and the quantities (T_s and K) taken from the fitted curve were unreliable as predictors.

Tom Mapp carried the curve-fitting comparison one step further. He tested four curves (a parabola, a trapezoid, a horizontal line, and the Rayleigh curve). The measure of comparison was the average squared (vertical) distance between the curve and the data points. Mapp used two techniques to find a best fitting curve for the Rayleigh equation and the parabola, the linear technique and a blind search. The blind search systematically sampled values from a bounded portion of the parameter space. The parameter set that yielded the best error measure became the center of a smaller "search box." When a new iteration failed to improve the error measure, the search was terminated. In every case the search technique produced a better fit than the linear method. The best curve, determined from the rank orderings of the final error measures for four projects, was the parabola. The study concluded that the Putnam model was successful at predicting milestones but did not fit the staffing data for our environment.

In this paper we analyze a new dynamic staffing model proposed by Parr [5]. To begin we review the general differences between static distribution models, based on a work breakdown structure, and dynamic distribution models, derived from a theory about problem solving. Then we examine the two theoretical (dynamic) models of Parr and Putnam to illuminate the critical assumptions that shape the curves and how they differ. Finally, we consider the claims made by dynamic staffing models and attempt to validate them using data from our environment. In particular, how well do dynamic models actually fit our data, and can the Parr model be used to predict project duration (in a manner similar to the Putnam model)?

DISTRIBUTION MODELS, CLAIMS, AND LIMITATIONS

Static distribution models start with a general description of the activities that constitute the software development process for a given environment. Then the tasks that comprise each activity are grouped under the right development phase. The important step is to distribute the total effort across these tasks. Each task is given a percentage, based on the skill and intuition of the model builder, and any available accounting data (assuming it reflects a similar environment of software methodol-

ogies). The percentages can be divided further to take into account different types of personnel (managers, analysts, programmers, or librarians) and different levels of experience that will be needed for the job. An example of a work breakdown structure is given by Wolverton [6]. When the functional specifications are complete, some adjustments will often be made in the baseline percentages to reflect the special demands of the particular project.

The static model provides a detailed staffing algorithm once an estimate is made for the total effort and the project schedule. A staffing algorithm is an excellent tool to monitor the progress of the project. First, the manager can use the algorithm to anticipate the fluctuations in his manpower needs before it becomes a problem. Because hiring new people is difficult, an increase in staffing requires some warning. In addition, the milestones of the schedule work like a sequence of checkpoints. When a milestone is not met, the work breakdown highlights which tasks are in trouble and possibly need more people.

The impulse to add more people to a late task is a natural one, but it can cause the task to be even later. It is unexpected phenomena like this that motivated the development of dynamic staffing models based on a theory of how we build software. Dynamic models propose assumptions to help explain such behavior. For example, adding more people to a working group increases the number of communication lines. The job of keeping people informed is more costly in terms of time and effort. Also, new people require an adjustment period, to get acquainted with the task, and will probably divert some of the energy of the original team members. On the other hand, there are inherent constraints in the software problem itself. A partial ordering of the individual subproblems exists, which limits the amount of work that can be done at the same time (and the number of people who can be effectively used). All these assumptions could help to explain why a part of the effort that is applied to a task does not result in any actual progress.

Dynamic distribution models are not alternatives to static staffing methods but instead complement them. Dynamic models deal with the kind of macroscopic quantities that are needed to use a static model. Used alone, dynamic methods lack the necessary detail to be an effective staffing algorithm, but they provide a glimpse of the overall picture. Dynamic models can estimate critical milestones in the schedule. They can serve as a means of checking the reasonableness of the percentages in the work breakdown structure with regard to the weekly effort expended. One aspect of the Putnam model even shows how a change in the specifications at any time will effect the schedule, the total effort, and the size of the code. Next we consider how a theoretical model derives an effort distribution.

SHAPING A DISTRIBUTION CURVE: THE THEORIES

The dynamic models rely on a set of assumptions describing how we build software. A software project consists of solving a bounded set of problems. Each problem represents some aspect of the design or implementation for which a decision must be made between possible alternatives. We are concerned with the constraints describing when effort can be effectively applied to solving these problems. The dynamic models of Parr and Putnam agree that the reason for a decrease in effort at the end of the project is an exhaustion of the problem set. This decrease reflects the nature of the debugging task. "Debugging is '99 percent complete' most of the time" [1, p. 154]. We do not have adequate measures to decide when a project is done, or even how much longer it will take. System debugging does not lend itself to people working in parallel, because errors tend to be discovered sequentially. The correction of one error uncovers another. It requires a small number of people working over an extended period to complete this phase of the project. Both models use an exponential tail to describe this situation.

The models disagree over what constrains the distribution curve at the start of the project. Putnam argues that progress can only be made once the development team becomes familiar with the problem and the proposed method of solution. The familiarization (or learning) rate that fit his data best was a straight line whose slope is determined by management's staffing decisions. However, there are practical limits as to how fast the buildup can be. First, it is hard to obtain new people, whether by hiring them or transferring them from other projects. Second, there is an organizational limit on the number of people that can communicate and work effectively with one another. The rate of the initial buildup also has implications for the duration of the project. Given a fixed amount of total effort, the faster the rise in staffing, the shorter the schedule. The size and the complexity of the problem fixes a minimum time period for the project duration.

Parr feels that these considerations focus on the wrong issue. It is important to understand how the problem itself limits the effort that can be effectively applied before considering those management decisions that are economically motivated. In that way we can examine an optimal staffing plan for the problem without concern for practical considerations, whose impact can be analyzed separately. Parr suggests that there are dependencies between the problems, so that the work

on a particular task cannot begin until others have been completed. These dependencies form a partial ordering. At any given time a subset of the unsolved problems exists, called the *visible* set, consisting of those that are ready to be worked on; in other words, all of the tasks on which a visible problem depends have been solved. The size of the visible set is the quantity that management is aware of, and (provided there are enough people to work on all of the visible problems at once) it should determine an optimal level of staffing.

The Rayleigh curve rises in a straight line from the origin to a rounded peak and then falls in an exponential tail. The formula for the Rayleigh curve is

$$y'(t) = 2Kate^{-at^2},$$

where

y' is the effort in man-hours expended per week,

t the time in weeks,

K the development effort in man-hours (the area under the curve), and

a a shaping parameter.

a determines the slope of the rising portion of the curve and equals $-\frac{1}{2}t_d^2$, where t_d is the point of maximum manning. When K represents the life-cycle cost of the system, t_d corresponds roughly to the development time up to acceptance testing. This equation makes explicit the inverse relationship between the learning rate and the project duration. In the discussion to follow K represents the development cost (that is, we assume no maintenance or enhancement).

The normalized Parr curve is bell shaped (symmetric about the origin) and trails off exponentially on both sides. The formula for the Parr curve is

$$y'(t) = \alpha K' Ae^{-\alpha\gamma t}/(1 + Ae^{\alpha\gamma t})^{1+1/\gamma},$$

where

y' is the effort in man-hours expended per week,

t the time in weeks,

K' the development effort in man-hours,

A the horizontal shift factor,

α the time normalization factor, and

γ a structuring index.

γ measures the extent to which formal structured techniques are a part of the development process. When $\gamma > 1$ the peak of the curve is skewed to the right. The purpose of structured programming is to delay implementation decisions through the use of abstraction and information hiding. These practices result in more time being spent in the specification and design phases so that the coding and testing phases will be simpler (partitioned in such a way as to minimize interfaces and allow maximum parallel effort). α stretches or shrinks the time variable onto a unitless scale, and A shifts the curve horizontally along the time axis.

K' has a different interpretation than K. Putnam assumed that each project has an official starting date prior to which no money or people are budgeted. This was reasonable in his environment, because a separate organization handled the preliminary work. If the starting date is $t = 0$, then the Rayleigh curve must pass through the origin. Parr argues that there is always some effort expended before the official project start. This early work serves the important function of defining the problem set that represents the desired software system (through feasibility studies and requirements analysis), establishing its internal structure (through functional specifications), and solving the top-level problems (through preliminary design) on which all the others depend. The positive effect of these activities is to expand the visible set of unsolved problems that is available to be worked on at the project start. General experience with the application area, specific design, or even code contributes to the structuring process. Thus the Parr curve does not pass through the origin. If K_0 represents the initial effort (the area under the curve before $t = 0$), then $K' = K + K_0$, and K_0, along with the shaping parameters α and A, determines $y'(0)$, the level of initial staffing. More will be said concerning this relationship in the section on estimating the Parr curve parameters.

A second difference between the two curves relates to the degree of flexibility in positioning the point of maximum staffing. As we mentioned previously, this point is determined by the slope of the initial rise (t_d being directly related to a). A large slope implies an early peak and a rapid fall in staffing. Conversely, a small slope implies a late peak with little or no decrease before acceptance testing. Basili and Zelkowitz comment that for medium-sized projects "the resource curve is mostly a step function." The Rayleigh curve seems inappropriate to this shape, and "variations in the basic curve so that it is flatter in its mid-range" are being investigated. The Parr curve is one possibility. In the next section we present our results from the curve fitting comparison between Parr and Rayleigh.

DO DYNAMIC MODELS ACTUALLY FIT OUR DATA?

We considered two paradigms in our analysis of curve fitting. First, we wanted to be able, given the data on the effort associated with a project, to tell what staffing algorithm (actually, what distribution curve) had been used in its development. The curve we were looking for

would fit the data better than the others. In particular, we set up a comparison between the two theoretical curves (those based on a software theory) of Parr and Putnam, as well as two control curves with reasonable characteristics (initial rise to a peak and then a fall). If a theoretical curve did better, then this would tend approximately to validate the assumptions made by the model.

The possibility remained that our data contained so much noise that none of the curves would stand out as better than the rest. In that case, a second paradigm is to be considered: Given the effort data and a staffing algorithm, we can decide whether in fact the algorithm had been used for the development process. This paradigm was tried with a staffing rule of thumb supplied by the contractor.

The noise comes from several sources. Since the data is weekly, weeks that contain holidays involve less total effort. If one member of the team is sick or on vacation, there is a drop in the weekly effort. If there is a problem, several people will work overtime and create a rise in the weekly effort. This is especially true when the average staff size is between 4.6 and 8.3 on the projects studied. To eliminate the noise we tried smoothing the data and combining four-week intervals. Unfortunately, this had little effect.

The nature of our effort data made it necessary to use an error measure for comparing curve fits; often a visual comparison was not possible. We chose the same measure as Mapp had used, namely the standard error

$$SE = \sum_{t=1}^{N} \frac{[f(t) - x(t)]^2}{N},$$

where

N is the number of data points,

$x(t)$ the effort in man-hours expended in week t (the data), and

$f(t)$ the distribution curve evaluated at t.

The technique to minimize SE involved two routines that were borrowed from the IMSL (International Math and Statistics Library) package. The first, ZXMIN, uses a quasi-Newtonian method to calculate the minimum of a user-supplied function. The routine requires an initial guess for the parameters of the function and then in an iterative fashion generates a new set of parameters from the old set. In order to avoid converging to a local minimum, we started the search at a large number of points. The second routine, XSRCH, did the selection of the initial parameter sets from a search box of reasonable values fixed by the user. There are two conditions that control the termination of the search process, the number of iterations and the differences between consecutive values of the parameters. One more twist was added to force the search to stay within the initial search box.

Table 2. An Initial Curve-Fitting Comparison: Rayleigh vs Parr Using SE

	Project				
	1	2	3	4	Average
Parr curve	939	2356	2204	2928	2204
Rayleigh curve	3379	4501	3926	4758	3926

In the first curve-fitting comparison the Parr curve did much better than the Rayleigh curve on each of four projects (see Table 2). The average of the standard error across the projects showed that the Parr fits were nearly twice as good (2204 to 3926). But these results were not very interesting, because the Parr curve has four parameters and the Rayleigh curve has only two. We had the suspicion that any curve with four parameters would have done better than one with two. In order to make our comparisons meaningful we decided to examine curves with an equal number of parameters. In the next test we therefore removed a parameter from the Parr curve and added one to the Rayleigh curve. We also included two more control curves. If a control did as well or better than the curves based on a theoretical distribution model, we could conclude that for our environment there was nothing special about the curve shapes.

The choice of parameter to remove from the Parr curve was decided once we noticed that the parameter A could change by several orders of magnitude and the curve still maintain a good fit (see Appendix A). The other parameters seemed to compensate in such a way as to suggest the power relationship

$$A = f(\alpha,\gamma) = 2^{100\alpha\gamma-2}.$$

A second possibility for removing A was to set it equal to a constant. For the projects in question the constant 1000 produced good fits. The results from comparing these three-parameter variations to the basic Parr curve (see Table 3) show that a constant A does almost as well as an extra parameter.

To increase the flexibility of the Rayleigh curve we

Table 3. Three- and Four-Parameter Fits

	Project				
	1	2	3	4	Average
$A = 1000$	939	2356	2913	3060	2317
$A = f(\alpha\gamma)$	991	7096	2704	3303	3524
Four-parameter curve	939	2356	2592	2928	2204

Table 4. Three-Parameter Curve Fit

	Project							
	1	2	3	4	5	6	7	Average
SE								
Parr	938	2356	2913	3060	2367	2072	2864	2367
Rayleigh	1837	3853	2517	3713	2980	2580	3350	3007
Parabola	1115	2298	2505	3497	3078	2508	3203	2601
Trapezoid	974	2265	2588	2981	2517	2722	3066	2445
Rank ordering								
Parr	1	3	4	2	1	1	1	13
Rayleigh	4	4	2	4	4	3	4	25
Parabola	3	2	1	3	3	2	3	17
Trapezoid	2	1	3	1	2	4	2	15

incorporated a horizontal shift factor so that the curve was not forced to pass through the origin. We borrowed the control curves from the Mapp study, a parabola and a "trapezoid" consisting of three straight lines. The exact formulation for these curves is given in Appendix B. The three-parameter curve fitting comparison (including graphs and tables of SE values and rankings) is presented in Table 4 and Figures 1–7. With respect to average SE values, the Rayleigh fits had improved, but Parr still did better. It is reassuring to note that the Parr curve also did better than either of the controls in terms of average fit and total rankings. However, we question whether the differences between these evaluations are large enough to be significant.

We concluded that the large fluctuations in the data for projects of this size effectively covered up the inherent shape of the effort distribution, if in fact such a shape exists as Putnam and Parr suggest. Consider our experience: When fitting the four-parameter Parr curve, the noise in the data allowed us to change the characteristics of the curve considerably (as reflected in the parameter A) and still retain a reasonable fit. Also, three very different curves, the trapezoid, the parabola, and Parr did equally well in the three-parameter comparison. The very data seem to contradict the assumption that there are constraints on staffing, whether introduced by management concerns or by problem dependencies. It therefore appears that we cannot tell what kind of software environment or staffing algorithm was used given only the effort data for a given project.

The second paradigm attempts to validate a staffing

Figure 1. Three-parameter fit of man-hours to weeks for project 1. Key: – – –, Parr curve; ···, Putnam curve; – · –, parabola; ———, trapezoid.

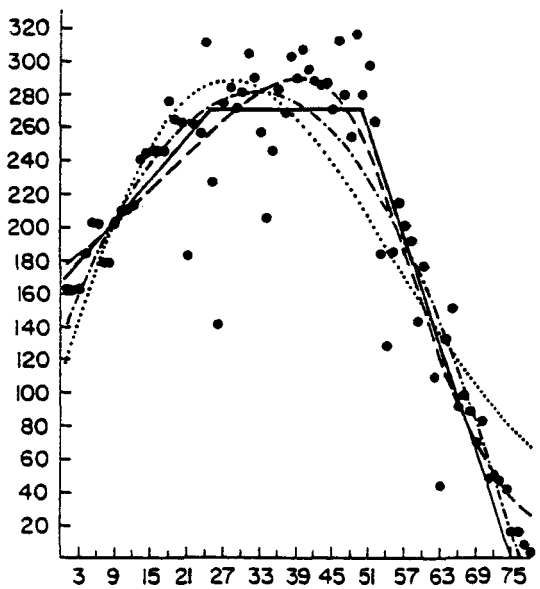
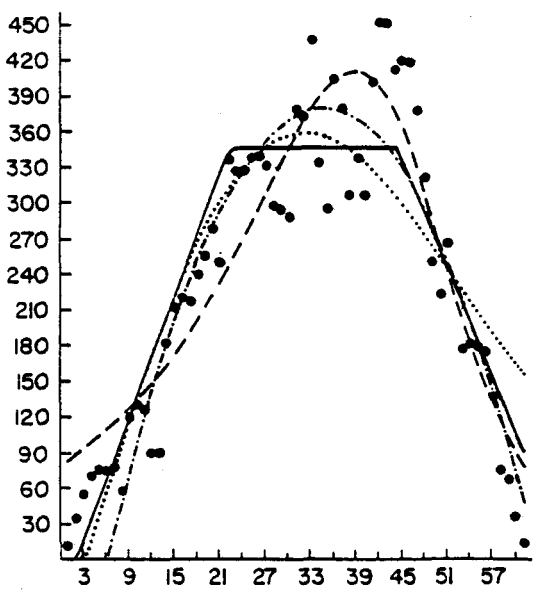


Figure 2. Three-parameter fit of man-hours to weeks for project 2. For key see Figure 1.



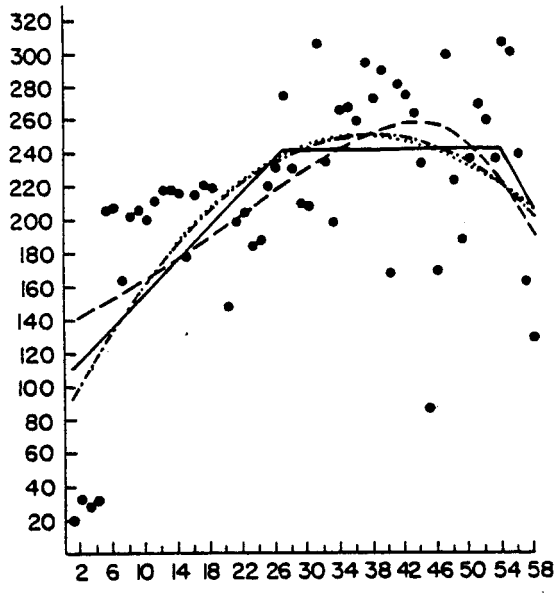


Figure 3. Three-parameter fit of man-hours to weeks for project 3. For key see Figure 1.

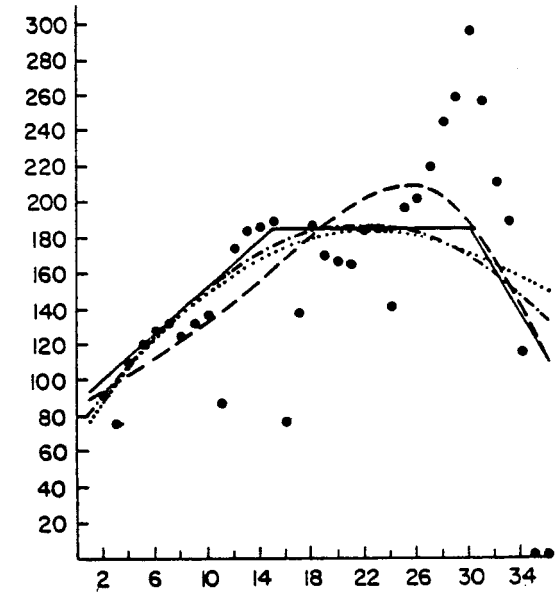


Figure 5. Three-parameter fit of man-hours to weeks for project 5. For key see Figure 1.

algorithm given both the algorithm and the effort data. The rule of thumb for staffing that the contractor tries to follow is this:

1. At the start of the project assign from one-half to three-quarters of full staffing (because of a lack of early funding and problems in finding available people).
2. At the end of the design phase, plus-or-minus a month, build to full staffing.
3. During the coding phase maintain full staffing.
4. During the testing phase, (a) if on schedule, decrease manning as appropriate; (b) if behind, work overtime; and (c) if there are late changes to the user requirements, increase manning by an additional one-third.

Figure 4. Three-parameter fit of man-hours to weeks for project 4. For key see Figure 1.

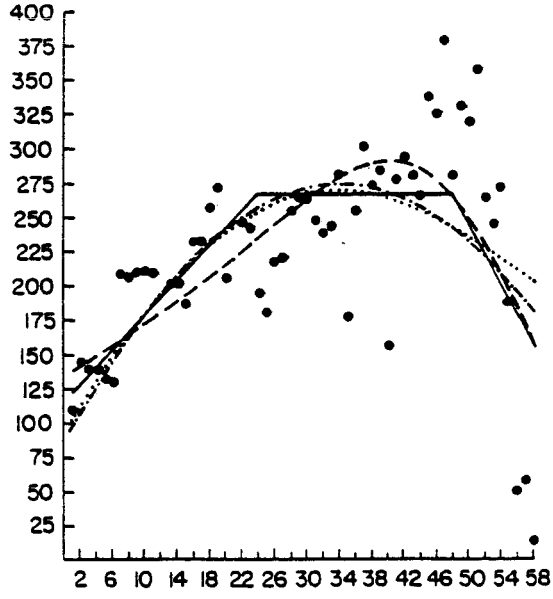
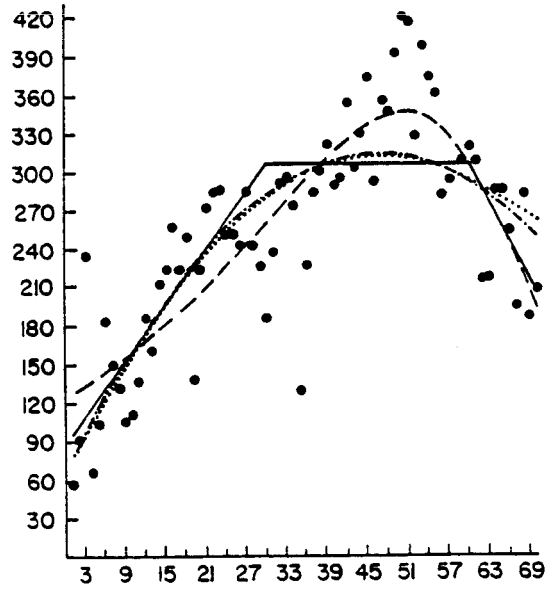


Figure 6. Three-parameter fit of man-hours to weeks for project 6. For key see Figure 1.



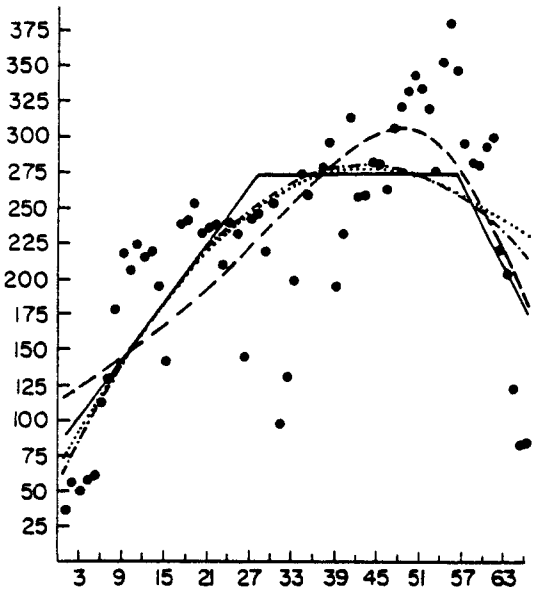


Figure 7. Three-parameter fit of man-hours to weeks for project 7. For key see Figure 1.

These guidelines convey the impression that management has considerable flexibility in terms of staffing to handle problems when they arise. The reason new people can be brought in at the end and contribute almost immediately is the similarity between the projects. Often a new system is a modification or enhancement of an old system, as seen in the percentage of existing code that gets reused, so little time is wasted in becoming familiar with a new system.

Next we wanted to check whether the contractor's rule of thumb was being used in practice. Since the algorithm is expressed as a step function, we needed to calculate averages for the phases concerned. In particular, we chose an 8-week period from the middle of each phase, which we thought could be representative in the sense that expenditures for those weeks took on roughly median values for the phase as a whole. We gave added weight to periods in which expenditures were more or less stable, whether the period fell in the middle or not. The averages computed from these periods are approximate. By selecting a different period the numbers can be changed by as much as 25 man-hours. Table 5 shows the numbers for five projects. If we assume that the numbers for the coding phase represent full staffing, they correspond fairly well to the algorithm. The average percentage of weekly design expenditures was 67% that of full staffing taken across all projects, a number midway within the range quoted in the algorithm for design, and various projects seemed to exhibit behavior that fit well into the three options for step 4. Project 1 decreased the level of staffing, projects 2 and 3 remained at the same level, and projects 4 and 5 increased staffing by one-third during the testing phase. The conclusion, then, is that we cannot reject the assumption that the contractor's algorithm is being used as a rough guideline by the managers. However, if we plotted the contractors algorithm as we did the other curves, the SE would be no better.

Table 5. Verifying the Contractor's Algorithm

Project	Design[a]	Code[a]	Test[a]	Design/code	Test/code
1	197	270	220	0.73	0.81
2	94	364	360	0.26	0.99
3	202	244	253	0.83	1.04
4	205	245	326	0.84	1.33
5	114	170	224	0.67	1.32
Average:				0.67	1.10

[a]Averages for 8-week periods (in man-hours per week).

REALITY AND THE PARR PARAMETERS

One of the benefits we mentioned in connection with theoretical effort distribution models was the ability to predict important milestones in the development schedule. However, before turning to the prediction problem, we wanted to be sure that the curves we fit to the effort data resulted in dates that were close to the actual milestones. This was another way of validating the model. In particular, we looked at the time period from the official start of the project through acceptance testing (t_a), or roughly the duration of the development activity. Solving the Parr equation for t_a yields

$$t_a = -\frac{1}{\alpha\gamma}\frac{\ln(K'/K'_a)^\gamma - 1}{1000},$$

where K'_a is the cumulative effort up to acceptance testing. (In the SEL environment we have estimated $K_a = 0.88K$. To convert these parameters into their Parr equivalents we added K_0 to each, so $K'_a = 0.88\,(K' - K_0) + K_0$.) The derivation of the equation for t_a is given in Appendix C, and the results of the comparison between the real values of t_a and the values taken from the curve are presented in Table 6. Except for project 1, the predicted values are within 5% of the true values. The bad estimate can be explained (at least in part) by

Table 6

	Project			
	1	2	3	4
t_a (estimated)	57.7	52.1	61.9	55.8
t_a (actual)	47.8	54.5	60.8	53.4

our not beginning to collect data for project 1 until well into the design phase.

Now that we had some confidence that the Parr curve fit the data about as well as any other and that the milestones calculated from the parameters were fairly accurate, we turned to the difficult problem of estimating the curve's parameters. K' is the total development effort, and a model like that of Walston and Felix [7] or Boehm [8] could be used to obtain an estimate. γ describes the degree to which formal (structured) methodologies are part of the development process. This parameter like Putnam's "technology constant," can be calibrated based on the techniques in use for a given environment. The remaining parameters A and α determine K_0, the amount of effort expended before the official start of the project; α converts the time variable onto a unitless scale, and A shifts the curve horizontally. Both depend on the duration of the project, the unit of measure for the time variable, and what part of the curve (how much of the exponential tails) is to be used to fit the data. α was introduced in the derivation of the Parr curve as a proportionality constant relating the rate of expending effort to the amount of work to be done at a given moment. If it took one person one time unit to solve each problem in the development effort, then α would be 1. It can be shown that when A is large (for our environment A was on the order of 1000) α is approximately equal to the y intercept of the distribution curve, $y'(0)$, divided by K_0 (see Appendix D). Our energies were therefore directed to finding a way to estimate these two quantities.

During the early stages of the project when the effort estimation and distribution models are needed, some initial effort data are available (Table 7). We attempted to use the data to estimate the y intercept of the distribution curve. Table 8 compares the y intercept with the first data point, the average of the first five data points, and the average of the first ten data points. For projects 1 and 4 the initial effort point is a close approximation to the y intercept, and after 5 weeks the estimate is even better. However, the averages of the initial effort for the other two projects do not begin to approximate the y intercept until after the tenth week.

This approach seemed to fail because of the nature of our data. Most projects have trouble finding enough people at the start, and many of the people who are assigned to the project begin by working part time, so even when the new effort allows a good deal of parallel activity at its inception, the problem of short staffing often squanders the opportunity for a fast start. As a result, the optimal manpower rates as reflected by the Parr curve are not met by the projects with early staffing problems. Using the initial effort data for a project is thus not an acceptable method of estimating $y'(0)$.

Table 7

	Project			
	1	2	3	4
Old code (%)	24.2	20.5	11.0	34.5
K_0/K'	40.6	8.9	33.5	28.0

Table 8

	Project			
	1	2	3	4
y intercept	179	79	138	136
Effort for first week	163	11	20	110
Average for first 5 weeks	175	49	64	133
Average for first 10 weeks	185	71	130	163

We also tried to estimate K_0, combining those activities that help define the problem before the official start of the project. Such activities as feasibility studies, requirements analysis, the use of existing design and code, and the general experience of the contractor with the application area partition the problem so that more people can work in parallel at an early stage in the development. For a rough estimate we chose the percentage of existing code because it was easy to get the data. Table 7 shows that comparison to K_0. Much of the variation in K_0 is not explained by this factor alone. To improve the comparison other factors (such as those mentioned above) will have to be incorporated into the estimate.

Thus we were unsuccessful in using the Parr curve as a predictor of such milestones as completion date since we were unable to associate the equation parameters with any data that would be easy to estimate at the onset of the project.

CONCLUSION

Dynamic distribution models offer an estimation tool for critical software quantities such as project duration, as well as a set of assumptions to enhance our understanding of problem solving behavior. To provide some assurance that these assumptions are valid for a given environment, we proposed fitting the effort distribution curve to actual data. In previous studies the Rayleigh curve proved to be a good method for estimating project duration, but for small- to medium-scale projects it did not fit the data. Thus there is some doubt whether the model can be used to monitor the expenditures of effort for an environment. This paper analyzed the applicability of an alternative curve developed by Parr. In a comparison of four curves with an equal number of pa-

Table 9. Parr Curve Fit for A Constrained (Project 1)

	<20	<25	<40	<100	<200	<1000
$100\alpha\gamma$	7.2	7.6	7.9	9.1	10.1	12.4
$f(\alpha\gamma)$	35.5	53.8	59.3	134.4	265.0	1296.1
SE	1382	1338	1218	1091	1019	939

Table 10. Parr Curve Fit for A Constrained (Project 4)

	<20	<50	<100	<500	<1000	<3000
$100\alpha\gamma$	5.9	8.1	9.1	11.7	12.9	14.9
$f(\alpha\gamma)$	14.8	69.1	133.4	831.7	1910.6	7643.4
SE	4123	3425	3341	3145	3060	2928

rameters, the Parr curve produced the best fits. However, the results and the data tend to contradict rather than support the theory on which the curve is based. The data imply that management has the ability to change staffing almost arbitrarily to meet the short-term needs of the project. The fluctuations in the data imply that a natural shape for the effort distribution may not exist for projects of this size.

The Parr model must do more than fit effort data. Although a fitted curve produced an accurate prediction of project duration, the crucial question is whether we can discover a way to estimate the Parr parameters themselves. Our efforts have not been fruitful up to this point, but other options of study remain. For now the Parr curve has limited usefulness as an effort distribution and resource estimation tool.

APPENDIX A. Eliminating a Parameter from the Parr Curve by a Power Relationship

The Parr curve is flexible enough to allow A to change by several orders of magnitude and still retain a reasonable fit. As A is increased, the product of α and γ increased in a similar way so as to suggest the following power relationship:

$$A = f(\alpha\gamma) = 2^{100\alpha\gamma - 2}.$$

This function was deduced by noticing the change in the parameters for various fits where the value of A was constrained to be less than some bound. In the cases of projects 1 and 4, the bounds were consistently set too low, so that by increasing the bound the fit improved. The results of contrasting A are shown in Tables 9 and 10.

APPENDIX B. Three-Parameter Curves

Three curves were compared with the Parr curve in the three-parameter test. The variable t is time measured in weeks.

For the Rayleigh (Putnam) curve,

$$y'(t) = 2Ka(t + t_0)e^{-a(t+t_0)^2},$$

where

y' is the effort in man-hours expended per week,

a a shaping parameter related to the time when the curve reaches a maximum,

K the total effort for the project in man-hours, and

t_0 a horizontal shift factor to remove the origin constraint.

For the parabola,

$$y'(t) = at^2 + bt + c \quad \text{if} \quad y'(t) > 0,$$
$$= 0 \quad \text{if} \quad y'(t) \le 0.$$

The parameters a, b, and c do not have any special meaning from an estimating point of view.

For the trapezoid,

$$y'(t) = [3(H - y_0)/T]t + y_0 \quad \text{if} \quad 0 \le t < T/3,$$
$$= H \quad \text{if} \quad T/3 \le t < 2T/3,$$
$$= 3H - (3H/T)t \quad \text{if} \quad 2T/3 \le t < T,$$
$$= 0 \quad \text{if} \quad t > T,$$

where

y' is the effort in man-hours expended each week,

H the maximum manning for the project in man-hours,

T an arbitrary time period in weeks, and

y_0 the manning at project start.

The measure for goodness of fit used in comparing the curves (see Table 4) was

$$\text{SE} = \sum_{i=1}^{N} \frac{[f(t) - x(t)]^2}{N},$$

where

N is the number of data points (the project duration in weeks),

$f(t)$ the value of the curve at t, and

$x(t)$ the actual effort in man-hours expended during week t.

APPENDIX C. Verifying the Time to Acceptance Testing as Predicted by the Parr Curve

The integral of the Parr curve is an equation for the cumulative effort expended up to time t. We solve this equation for the time t_a, the time to acceptance testing.

$$Y'(t) = K'(1 + Ae^{-\alpha\gamma t})^{-1/\gamma}$$

Substitute $Y'(t_a) = K_a'$ and $A = 1000$:

$$K_a/K' = (1 + 1000e^{-\alpha\gamma t_a})^{-1/\gamma},$$
$$1000e^{-\alpha\gamma t_a} = (K'/K_a)^{\gamma} - 1,$$
$$-\alpha\gamma t_a = \frac{\ln[(K'/K_a)^{\gamma} - 1]}{1000},$$
$$t_a = -\frac{1}{\alpha\gamma}\frac{\ln(K'/K_a)^{\gamma} - 1}{1000}.$$

Table 6 compares the estimate of t_a using a fitted three-parameter Parr curve with the actual data (the values for t_a are in weeks).

APPENDIX D. Searching for a Physical Interpretation of the Parameter α

α can be expressed in terms of $y'(0)$ and K_0. The three-parameter Parr curve evaluated at $t = 0$ is

$$y'(0) = 1000\alpha K'/1001^{1+1/\gamma}.$$

Using the cumulative distribution curve, the initial effort K_0 is

$$K_0 = Y'(0) = K'1001^{-1/\gamma}.$$

Solving for K' and substituting back into the first equation leaves $1000\alpha K_0/1001$. α is approximately equal to $y'(0)/K_0$.

Table 6 shows a comparison of the percentage of existing code that is reused in the new system to the percentage of initial effort as computed by the Parr curve. Table 7 compares the effort data at the beginning of the projects with the y intercept of the three-parameter Parr curve. Three measures for the effort data are used: the first data point, the average of the first five data points, and the average of the first ten. All the numbers are in units of man-hours per week.

REFERENCES

1. F. Brooks, *The Mythical Man-Month; Essays on Software Engineering*, Addison-Wesley, Reading, Massachusetts, 1975
2. L. Putnam, A General Empirical Solution to the Macro Software Sizing and Estimating Problem, *IEEE Trans. Software Eng.* 4 (4) (1978).
3. V. Basili and M. Zelkowitz, Analyzing Medium-Scale Software Development, *Proc. 3rd Int. Conf. Software Eng.*, Atlanta, Georgia, May 1978
4. T. Mapp, Applicability of the Rayleigh Curve to the SEL Environment, unpublished, University of Maryland, 1978
5. F. Parr, An Alternative to the Rayleigh Curve Model for Software Development Effort, *IEEE Trans. Software Eng.* (May 1980).
6. R. Wolverton, The Cost of Developing Large Scale Software, *IEEE Trans. Comput.* 23 (6) (1974).
7. C. Walston and C. Felix, A Method of Programming Measurement and Estimation, *IBM Syst. J.* 16 (1) (1977).
8. B. Boehm, *Software Economics*, Prentice Hall, Englewood Cliffs, N.J. 1981.

SECTION 4 – SOFTWARE MEASURES

SECTION 4 - SOFTWARE MEASURES

The technical papers included in this section were originally published as indicated below:

- Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10, copyright 1979 Pergamon Press (reprinted by permission of the publisher)
- Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1, copyright 1981 Elsevier-North Holland (reprinted by permission of the publisher)
- Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981 (reprinted by permission of the authors)

D6-61

Computers & Structures Vol. 10, pp. 39–43
0045-7949/79/0401-0039/$02.00/0

80022

MEASURING SOFTWARE DEVELOPMENT CHARACTERISTICS IN THE LOCAL ENVIRONMENT†

VICTOR R. BASILI and MARVIN V. ZELKOWITZ

Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.

(*Received* 12 *May* 1978)

Abstract—This paper discusses the characterization and analysis facilities being performed by the Software Engineering Laboratory which can be done with minimal effort on many projects. Some examples are given of the kinds of analyses that can be done to aid in managing, understanding and characterizing the development of software in a production environment.

INTRODUCTION

Software development is big business. Estimates on the actual expenditures for software development and maintenance were ten billion dollars in 1973[1] and most likely 15–25 billion dollars today. These are only estimates because little data is gathered by the software industry in monitoring itself, analyzing its environment and defining its terms.

The software product and its development/maintenance environments cover a wide range. The product varies from first time, one of a kind systems, to standard multi-level run of the mill systems; from large scale hundreds of man–year developments to small scale one to two man–year developments. The environment varies from shops dedicated to the development of software to organizations which simply maintain their existing software system. A large number of methodologies, tools and techniques are available to help in the cost effective production of maintainable software. However, most of these techniques involve tradeoffs when applied in actual practice; some tools are impractical in certain environments and some techniques may not be applicable in other environments.

For example, for a new one-of-a-kind project where some specifications are still unknown or subject to change (not a recommended procedure), incremental development techniques, such as iterative enhancement[2] may be more cost effective than the more standard top down approach. Some tools, such as requirements analyzers[3] which are highly effective in the development of large scale systems, are not effective when the project is relatively small due to the substantial overhead in using the tool. Peer code reading is impossible in an environment of only one programmer.

Understanding the characteristics of a particular software environment leads to more cost effective maintainable software. This requires knowledge of the various parameters that affect the development of software and its maintenance. Unfortunately there is little effort expended in analyzing this process in local environments. Most of the data has come from the very large scale developments, projects like OS360, Sage, Gemini and Saturn[4].

Although these projects are major contributors to the software development budgets, they are not necessarily typical of software development across the industry. However, they are easiest to secure funding for collecting data and analyzing it. For example, if the budget for a project is twenty million dollars, then it is easy to add two hundred thousand for data collection and analysis, a mere 1% overhead. However, if the project has a budget of two hundred thousand dollars, then adding fifty thousand for data collection imposes a prohibitive 25% overhead.

What characterizes these large scale software development projects? The development activities usually involve about 30% analysis and design, 20% coding and 50% testing. However, development costs account for only 20% of total system costs on some projects if maintenance and modification activities are included[1].

These cost characteristics however are different for different software environments. What characterizes the projects studied above is that they are large one time only systems. Testing is very expensive because it is difficult to integrate the various pieces of the system into a working unit. Clearly smaller better understood systems would require a smaller proportion of the testing time and possibly less design and analysis time.

The authors have been analyzing development in an environment in which the software is of the six–ten man–year variety involving the development of ground support software for spacecraft control; a set of problems whose basic solutions and designs are fairly well understood. Thus the tailoring of methodoligies and tools for this environment would surely be different than in other environments.

THE SOFTWARE ENGINEERING LABORATORY

The Software Engineering Laboratory began in August, 1976 to evaluate various techniques and methodoligies to recommend better ways to develop software within the local NASA environment. Three groups participate in the Laboratory—the University of Maryland, whose role is to develop an operational measurement environment and analyze the development process; NASA Goddard Space Flight Center, whose role is to implement the operational measurement environment and whose goal is to discover ways to develop more product for the money spent; and the contractor, Computer Sciences Corporation, whose role is to supply data as they develop software and whose goal is to gain feedback on project development both for understanding

†Research supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland.

the characteristics of past development and to monitor software development in real time.

More specifically, the goals of the Laboratory are:

1. Organize a data bank of information to classify projects and the environment in which they were developed.
2. Compare what is happening with what is supposed to be happening (e.g. are the proposed methodologies being employed as they are supposed to be implemented?).
3. Isolate the significant parameters that characterize the product and the environment.
4. Test out existing measures and models as they appear in the literature (usually for large scale software developments) and develop measures for the local environment.
5. Analyze methodologies and their instrumentation in the local environment.
6. Discover and recommend appropriate milestones, methodologies, tools and techniques for use under given conditions in order to develop more manageable, maintainable, reliable, and less expensive software products.

The research objectives of the Laboratory can be divided into three basic areas: management, reliability and complexity. The *management* study is to analyze and classify projects based on management parameters, and investigate management measures and forecasting models. The *reliability* study is to examine the nature and causes of errors in the environment, find classification schemes for errors and expose techniques that reduce the errors that occur in the local environment. The purpose of the *complexity* study is to gain insight into the nature of complexity and develop models that correlate well with those insights and discover whether various techniques create more systematic and thus easier to maintain program structures.

The primary data gathering technique for the Laboratory is a set of seven reporting forms:

General Project Summary

This form is used to classify the project and is used in conjunction with the other reporting forms to measure estimated vs actual project progress. It is filled out by the project manager at the start of the project, at each major milestone, and at completion. The final report should accurately describe the system development life cycle.

Programmer/Analyst Survey

This form is filled out by each programmer at the start of the project, and is used to classify the background of all project personnel.

Component Summary

This form is used to keep track of the components of a system. A component is a piece of the system identified by name or common function (e.g. entry in a tree chart, COMMON block, subroutine). With the information on this form combined with the information on the component status report, the structure and status of the system and its development can be monitored. This form is filled out for each component at the time that the component is identified (usually during the design stage), and at the time it is completed (usually during testing). It is filled out by the person responsible for the component.

Component Status Report

This form is used to keep track of the development of each component of the system. The form is turned in weekly by each person on the project, and it identifies the components worked on, hours devoted to each component, and tasks performed (e.g. design, code, review).

Resource Summary

This form keeps track of project costs on a weekly summary basis. It is filled out by the project manager and lists for all personnel the total number of hours spent on the project.

Change Report

The change report form is filled out every time the system changes because of change or error in design, code, specifications or requirements. The form identifies the error, its cause and other facets of the project that are affected.

Computer Program Run Analysis

This form is used to monitor computer activities used in the project. Entries are made every time a run is submitted for processing. The form briefly describes the purpose of the run (e.g. compile, test, file utility), and the results (e.g. successful, error termination with message).

DATA COLLECTION ON A SMALLER SCALE

The research goals of the Software Engineering Laboratory require the collection of large amounts of data to make full investigations into the nature of the software development process. The information being collected by the Laboratory, due to its research nature, is ambitious and not cost effective for simple management control; it requires a major expenditure just for processing and validating data for inclusion into the data base.

However, it is possible to gather less data to get effective results in analyzing the characteristics of the local software environment. For example, a subset of the information contained essentially on only three basic forms is used for the analysis in the next section. The three forms are the General Project Summary, the Resource Summary and the Change Report forms.

From the General Project Summary the following information is used:

1. Project description including the form of input (specifications), products developed and products delivered.
2. Resources of computer time and personnel, including constraints and usable items from similar projects.
3. Time including start and end dates and estimated system lifetimes.
4. Size of project including various measures such as lines of code, source lines and number of modules.
5. Cost estimates, man–month estimates and schedules.
6. Organization factors, personnel and the kinds of people used (e.g. managers, librarians, programmers).
7. Methodologies, tools and techniques used.

Data from the Resource summary includes weekly charges for manpower and computer time, and other costs for all categories of personnel. The change report form supplies data on changes made to the system, when they were made, what modules were affected by the change, and why the change was made.

PROGRESS FORECASTING

One important aspect of project control is the validation of projected costs and schedules. A model of esti-

mating project progress has been developed and with it estimates on project costs can be predicted.

The Rayleigh curve has been found to closely resemble the life cycle costs on large scale software projects[5, 6]. The curve yielding current resource expenditures (y) at time (t) is given by the equation:

$$y = 2K\,a\,t\exp(-a\,t^2)$$

where the constant K is the total project cost, and the constant a is equal to 1/(Td**2) where Td is the time when development expenditures reach a maximum. The following analysis demonstrates how this data can be used for management control of a project. The data was obtained on projects built for NASA and monitored by the Software Engineering Laboratory.

For each project in the NASA environment, requirements analysis yields estimates of the total resources and development time needed for completion, which is recorded on the General Project Summary form. The following three parameters are relevant to this analysis:

1. Ka, total estimated resources estimated to be needed to complete the project through acceptance testing (in hours).
2. Yd, the maximum resources estimated to be needed per week to complete the project (in hours).
3. Ta, the number of weeks estimated until acceptance testing.

Since the Rayleigh curve has only two parameters (K and a), the above system is over specified and one of the above values from the General Project Summary can be determined from the other two. Thus the consistency of those estimates can be validated. Alternatively, by estimating two of these parameters (e.g. total cost and maximum weekly expenditures), then the third value (e.g. completion date) can be calculated.

For example, since budgets are generally fixed in advance, there is usually little freedom with total resources available (K). Also, since a fixed number of individuals are usually assigned to work on the project, the maximum resources Yd (at least for several months) is also relatively fixed. Therefore, the completion date (Ta) will vary depending upon K and Yd.

As stated above, Ka is the total estimated resources needed to develop and test the system through the acceptance testing stage. For each environment, the actual resources K must be obtained from this figure. There are several methods for estimating K. One approach is by the empirical data available on past projects. By studying past projects as NASA, this figure is 12% greater than estimated expenditures (hence K = Ka/.88). The remaining 12% is for last minute changes after acceptance testing. Since maintenance costs are not covered, this figure seems quite low when compared to other programming environments—the corresponding figure in other organizations that do include maintenance costs will probably be correspondingly higher.

Give K, a was computed by assuming different values of Td to yield the given value of Yd on the General Project Summary. Then given constant a, the estimated date of acceptance testing Ta can be comput as follows:

The integral form of the Rayleigh curve is given by:

$$E = K(1 - \exp(-at^2))$$

where E is the total expenditures up to time t. From the previous disucssion, we know that at acceptance testing, E is .88 K (for NASA). Therefore,

$$.88\,K = K(1 - \exp(-at^2)).$$

Solving for t yields:

$$t = \mathrm{sqrt}(-\ln(.12)/a).$$

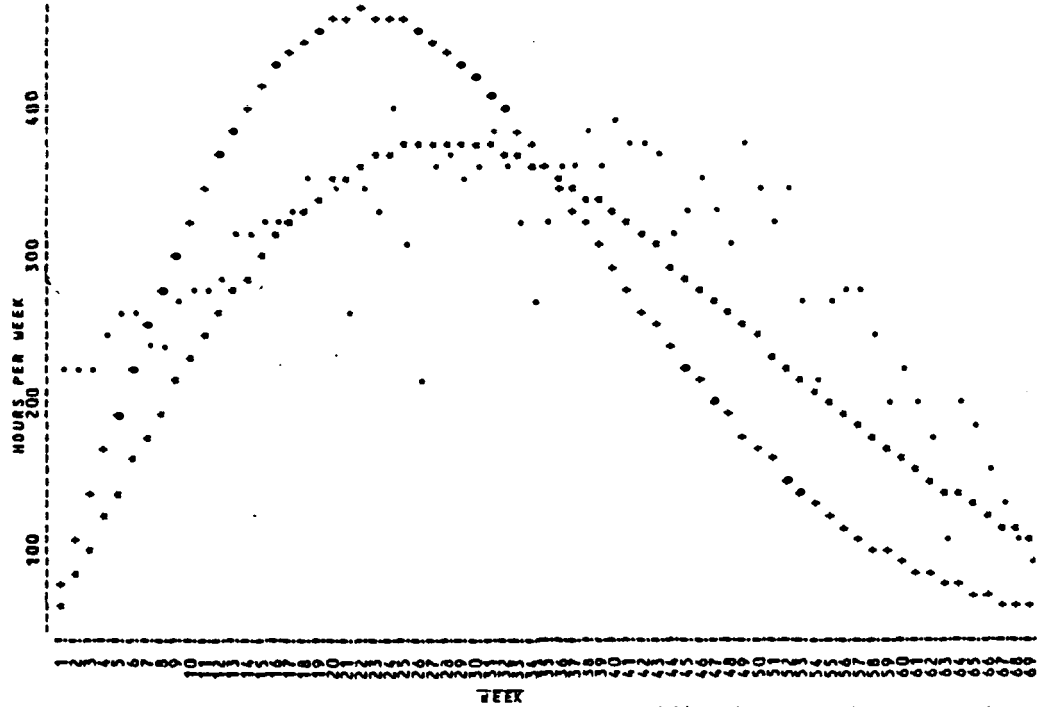


Fig. 1. Project A—Estimated resource expenditures curve based upon initial estimates. *. Estimating curve with Yd (maximum resources) fixed. +. Estimating curve with Ta (Completion date) fixed. ·. Actual data.

Also, in a second analysis, the estimated acceptance time Ta was fixed in order to yield a value of a (and hence Yd) that represents the manpower needed to finish on schedule.

If the original estimates from the General Project Summary are accurate, then the estimated and calculated values should be comparable. If the maximum manpower estimate was reasonable, then the predicted date for acceptance testing should be similar to the estimated date on the General Project Summary. If this acceptance date is reasonable, then maximum manpower estimates should be similar to the calculated values.

Figure 1 represents data from one actual project. According to the above analysis two different Rayleigh curve estimates were plotted. The curve limiting maximum weekly expendiutres (Yd) might be considered the more valuable of the two since it more closely approximates project development during the early stages of the project. In this case, the weekly expenditures from the General Project Summary were insufficient for completing acceptance testing by the initially estimated completion date Ta. The model predicted acceptance testing in 58 weeks instead of the proposed 46 weeks. The actual date was 62 weeks—yielding only a 7% error (Fig. 2).

In order to complete the project in 46 weeks, up to 440 hr per week (rather than the estimated 350 hr per week) would have to be spent.

As it turned out, the project used approximately 1600 hr more than initially estimated and maximum weekly resources were slightly more than original estimates (371 hr/week instead of 350 hr/week). If these corrected figures for Ka and Yd are used in the analysis, then Ta, the date for acceptance testing, is 60 weeks instead of the actual 62 weeks—an error of only 3%.

OVERHEAD

Overhead is often an elusive item to pin down. In our projects three aspects of development have been identified: programmer effort, project management, and support (librarians, typing, etc). In one project (Fig. 3), programmers accounted for about 80% of total expenditures with the support activities taking about one third of the remainder. In addition, only about 60% of all programmer time was accountable to explicit components of the system (as reported on the Component Status Report). The remaining time includes activites like meeting, travel, training sessions, and other activities not directly accountable. This "loss" of time is a signficant overhead item which must be considered in developing accurate project budgets.

ERROR ANALYSIS

The correction of errors in a system is the major task of integration testing. Even a simple counting of errors can be useful as a management estimating tool. Figure 4(a) represents the number of error reports reported per week on one NASA project. It remained surprisingly constant over the testing stage. However, the more interesting measure is the *handling rate* [7], or the number of different components altered each week (Fig. 4b).

Consider the following set of assumptions:

1. The number of errors in a system is finite, but unknown.
2. The probability of finding an error is proportional to the number of individuals working on the problem.
3. The probability of finding an error is random and uniformly distributed.

These three assumptions lead to a Poisson distribution

$$y = e^{-at}$$

as the probability of an error remaining after time t. Furthermore, if we include the assumption that the probability of fixing a found error (as opposed to creating a new error by fixing the previous error) is the function a = bt (e.g. errors are "easier" to find as you get "good at it"), then the resulting distribution is the same Rayleigh curve described previously [5].

Therefore, if N is the total number of errors in a system, and if h is a measure of the maximum number of errors found per week, then the number of errors found per week agrees with the curve:

$$y = 2Nht \exp(-ht^2).$$

A preliminary evaluation of the data of Fig. 4 (and other projects) seems to bear out these assumptions. Therefore, by using least squares techniques, the following algorithm can be used to measure testing progress:

1. Collect data on errors reported for several weeks.
2. Use least squares to fix a curve to this data. This gives a measure of N (modules handled) and h (a measure of maximum errors found).
3. N gives the number of modules in error in the system, however, this value can never be reached

INITIAL ESTIMATES FROM GENERAL PROJECT SUMMARY	
Ka, Resources needed (hours)	14,213
Ta, Time to completion (weeks)	46
Yd, Maximum resources/week (hrs)	350
COMPLETION ESTIMATES USING RAYLEIGH CURVE	
K, Resources needed (hours)	16,151
Estimated Yd with Ta fixed (hrs)	440
Estimated Ta with Yd fixed (hrs)	58
ACTUAL PROJECT DATA	
K, Resources needed (hrs)	17,742
Yd, Maximum resources (hrs)	371
Ta, Completion time (weeks)	62
Ta, estimated using actual values of K and Yd (weeks)	60

Fig. 2. Estimating Ta and Yd from General Project Summary data.

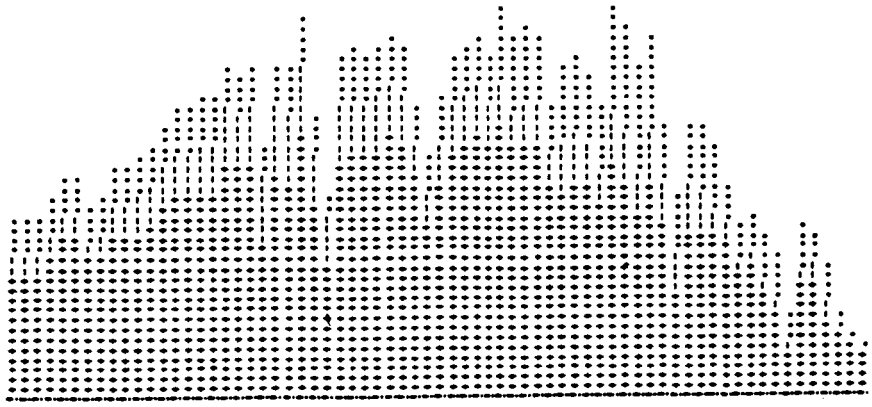
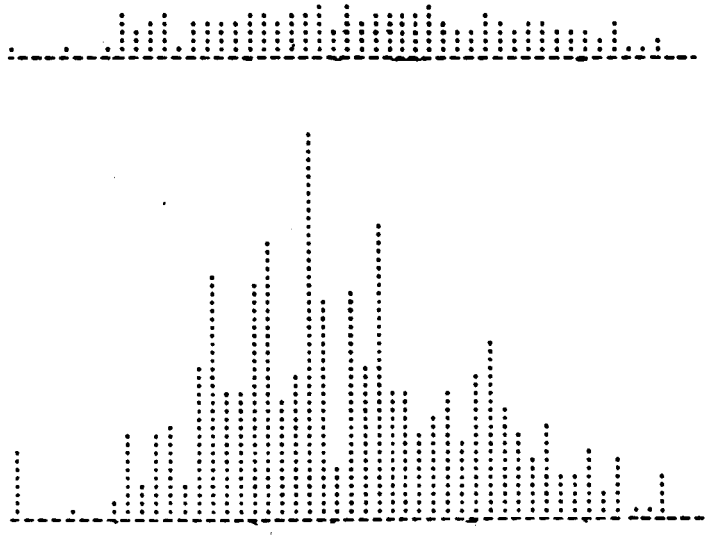


Fig. 3. Resources expended on various developmental activites. +, Programmer effort. −, Management effort. ·, Support effort (librarians, typing, clerical, etc.).

(a)



(b)

Fig. 4. Handling and report rates on one project. (a) Report rate by week. (b) Handling rate by week.

exactly. Compute the time needed to get the number of remaining errors to an "acceptable" level[8].

The project represented by Fig. 4 shows the practicality of this measure. This project has a total of 1115 components that were handled. A least squares fit yielded an N of 1024.9 and an h of .0009024 with a correlation of .7264. This figure of 1024 was only an error of 8% in the true handling rate. Current research is studying this aspect of errors in order to refine this measure further.

Acknowledgements—We would like to acknowledge the contributions and cooperation of Mr. Frank McGarry, head of the Systems Development Section of NASA Goddard Space Flight Center. He has been instrumental in organizing the Laboratory and in interfacting with the contractor in order to see that the data is collected reliably and timely. We would also like to thank Computer Sciences Corporation for their patience during form development and their contributions to the organization and operation of the Laboratory.

REFERENCES

1. B. Boehm, Software and its impact: a quantitative assessment *Datamation* 97–103 (July 1977).
2. V. Basili and A. J. Turner, Iterative enhancement: a practical technique for software development. *IEEE Transactions Software Engng* 1(4), 390–396 (1975).
3. D. Teichroew and E. A. Hershey, PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions Software Engng* 3(1), 41–48 (1977).
4. R. Wolverton, The cost of developing large scale software. *IEEE Transactions Comput.* 23(6), 615–636 (June 1974).
5. P. Norden, Use tools for project management. *Management of Production.* (Edited by M. K. Starr) pp. 71–101. Penguin Books, Baltimore, Maryland (1970).
6. L. Putnam, A macro-estimating methodology for software development. *IEEE Computer Society Compcon*, pp. 138–143, Washington, D.C. (Sept. 1976).
7. L. A. Belady and M. M. Lehman, A model of large program development. *IBM Systems J.* 15(3), 225–252 (1976).
8. J. D. Musa, A theory of software reliablility and its application. *IEEE Transactions Software Engng* 1(3), 312–327 (Sept. 1975).

Programming Measurement and Estimation in the Software Engineering Laboratory*

Victor R. Basili
Department of Computer Science, University of Maryland

Karl Freburger
General Electric Information Services

This paper presents an attempt to examine a set of basic relationships among various software development variables, such as size, effort, project duration, staff size, and productivity. These variables are plotted against each other for 15 Software Engineering Laboratory projects that were developed for NASA/Goddard Space Flight Center by Computer Sciences Corporation. Certain relationships are derived in the form of equations, and these equations are compared with a set derived by Walston and Felix for IBM Federal Systems Division project data. Although the equations do not have the same coefficients, they are seen to have similar exponents. In fact, the Software Engineering Laboratory equations tend to be within one standard error of estimate of the IBM equations.

INTRODUCTION

Many models of software development have been proposed in the literature. They all assume some set of relationships among the factors affecting the process. One of the goals of the Software Engineering Laboratory (SEL) [1, 2] has been to try to understand the development process by collecting and using data to evaluate the relationships proposed in the various models.

The Software Engineering Laboratory is a joint effort of NASA/Goddard Space Flight Center, Computer Sciences Corporation, and the University of Maryland. Its general goals have been to analyze the development of software in order to evaluate software development practices, models, and metrics so they may be better applied in understanding, managing, and engineering the process and the product. This paper analyzes one particular set of programming factors and their interrelationships. It completes a previous study [3] based on fewer data. These factors include the development effort, lines of code, number of modules, duration, pages of documentation, team size, and productivity. One of the most interesting relationships is between lines of code and effort. Contrary to intuition, previous researchers have reported that the relationship between these two factors is almost linear [4, 5]. Several studies have been conducted on a subset of these relationships.

Chrysler [6] collected data on 36 programs in one organization, Johnson [7] collected data on 169 programs in one organization, and LaBolle [8] and Nelson [9] analyzed data derived largely from the System Development Corporation. Jeffrey and Lawrence [4] present results obtained from the analysis of 103 programs from three organizations. Walston and Felix [5] collected data on 60 projects in one organization. A full discussion of previous programming productivity research may be found in the article by Chrysler [6].

It is clear that because of biases in the data and data collection process and the lack of control in the various studies, including our own, only the collection of data in many environments by many researchers will permit a wealth of evidence to be assembled sufficient to generate confidence in the relationships derived. In order to do this, however, results must be published using agreed upon, well-defined terms and explicitly stated environmental constraints so that the experimenter can relate what he is testing to previous studies.

To evaluate the relationships between factors, we

*This research was supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

Address correspondence to V. R. Basili, Department of Computer Science, University of Maryland, College Park, Maryland 20742.

The Journal of Systems and Software 2, 47–57 (1981)

tried to compare our findings with those of a previous study using the same definitions and trying, whenever possible, to make clear the differences in the two environments. Some variations of these factors were also studied and these are explicitly defined in this paper. The data here were obtained from a set of projects developed at NASA/Goddard by Computer Sciences Corporation. The findings are compared with the results of a study by Walston and Felix [5] at IBM Federal Systems Division. The major differences in the two environments are summarized below.

The SEL data base currently includes 15 projects, ranging in size from 1.6 to 112 thousand lines of code and in effort from 1.8 to 116 man-months. The Walston-Felix IBM data base includes 60 projects ranging in size from 4 to 467 thousand lines of code and in effort from 12 to 11,758 man-months. The IBM project data involve eight different languages on 66 different computers covering a very wide range of applications, personnel, and experience. The SEL projects are all in FORTRAN on two different computers and involve predominantly the functions for ground support software for satellites. Most projects were developed from a reasonably common programming pool, and most of the designs were well understood and similar to work previously performed, if not by the particular individual at least by the organization. In fact, most of the top-level design is somewhat standard.

Further discussion of the SEL data is given in the next section. The third section discusses the basic relationships derived from SEL using the same techniques as those of Walston and Felix [5]. Our data are then fit to the Waltson–Felix equations where appropriate. We conclude by attempting to validate the hypothesis, suggested by Jeffery and Lawrence [4], of a linear relationship between effort and product size.

THE SAMPLE DATA

Fifteen completed projects were used in this study. All were developed at NASA's Goddard Space Flight Center by NASA personnel and outside contractors. Five of the 15 projects are attitude determination systems, developed on an IBM System 360 in FORTRAN with some minor assembly language code. One project is an attitude determination support utility used to calculate parameters needed by the larger attitude determination systems. It was a one-man project developed on a PDP 11/70 and converted to an IBM System 360, which was the operational machine. The seventh project is an interactive graphics package developed on a PDP 11/70 in FORTRAN and MACRO-11 assembly language. A similar system already existed on an IBM System 360. The other eight data points are separately developed subsystems of a single attitude determination system. The individual data points from design through testing represent their independent development profiles and do not include any subsystem integration effort. They were developed for an IBM System 360 in FORTRAN. Each subsystem was developed predominantly by a single individual. More information on several of the projects, methodologies, and environment are given by Basili et al. [1].

Data were collected from these projects with the forms and techniques described by Basili et al. [1]. The data of interest in this study were the total effort required to produce the finished product, lines of source code in the finished product, number of modules, project duration, documentation size, productivity, and average staff size. A detailed description of these follows:

a. The *total effort E* is defined as the number of man-months of effort used on a project, starting when the requirements and specifications become final through acceptance testing. It includes programming effort and managerial and clerical overhead. One man-month of effort is defined here as $173\frac{1}{3}$ man-hours.
b. The total number of *delivered lines L* of source code is defined as the total number of lines of source code delivered as the final product (expressed in thousands of lines). It does not include stubs or any code thrown away. Source lines are 80-character source records provided as input to a language processor, including data definitions and comment lines.
c. The number of *lines of new code* (NL) is the number of source lines in the final product that are not reused code (expressed in thousands of lines). A block of code is considered to be reused if it was developed for a different project and less than 20% of the code is changed; if more than 20% of the code is changed, the whole block is considered new code.
d. The number of *developed lines* of code (DL) is a derived quantity equal to the number of new lines plus 20% of the reused lines: $DL = NL + 20\%(L - NL)$. The 20% overhead is charged to account for such items as system integration and full system test.
e. The total number of *modules M* in a project is the number of modules delivered in the final product. For all of the projects in this study, a module is defined as a separately compilable entity, such as a subroutine, function, or BLOCK DATA unit.
f. The number of *new modules* (NM) is the number of modules in the final product that are not reused modules. A module is considered to be reused if it was developed for another project and has less than 20% of its code changed.

g. *Project duration D* is defined to be the time (in months) from the start of a project (receipt of requirements and specifications) to the end of acceptance testing.
h. *Documentation* (DOC) is measured in pages and is defined as the program design, test plans, user's guide, system description, and module descriptions. The program design is a handwritten document. The module descriptions contain a one-page description of each module in the final product.
i. *Productivity P* is a derived quantity, defined as the ratio L/E of total lines of source code to the total effort required to produce the lines of code. Productivity is expressed in lines of code per man month of effort.
j. The *average staff size S* of a project is defined as the total man months of effort divided by the project duration: $S = E/D$.

Because one of the stated objectives of this research was to compare the results with those of a previous study [5], some of the definitions were chosen to be consistent with the definitions used in that study. The definitions of new and developed source lines of code (NL and DL) were selected to match the definitions used in the programming environment under study. The definition of documentation size was constrained by the data available.

ANALYSIS AND RESULTS

This section presents an analysis of various relationships that may be useful as estimating aids to project personnel. The data used for each variable are as described in the previous section. A summary of the results is presented in Table 1 (see also the end of this paper).

Where Walston and Felix performed an analysis of a similar relationship a comparison of the results is given. Where the results of this study and the Walston–Felix study are considerably different, an attempt is made to determine what factors (if any) may contribute to that difference.

To compute the relationships between the variables, two-variable regression is used. For exponential relationships (such as those presented in the Walston–Felix paper), the data are first linearized by taking logarithms. A two-variable linear regression (least squares fit) is then performed on the transformed data. The linear coefficients become exponential relationships when transformed back into the original domain of the data. For linear relationships, a two-variable linear regression is performed on the data.

The standard error of estimate (SE) provides an estimate of the range above and below the line of estimation within which a certain proportion of the items may be expected to fall if the scatter is normal. Assum-

Table 1. Summary of Results

Estimated variable	SEL equation	SE[a]	r**2[b]	Level of significance	Walston–Felix equation	SE[a]	r**2[b]
Total effort	E = 1.38*(L**0.93)	1.41	0.93	0.001	E = 5.2*(L**0.91)	2.51	0.64
	E = 1.58*(NL**0.99)	1.31	0.96	0.001			
	E = 1.48*(DL**0.98)	1.29	0.96	0.001			
	E = 0.652*(M**1.19)	1.49	0.90	0.001			
	E = 0.183*(NM**1.05)	1.57	0.87	0.001			
	E = 1.04*L + 2.04	16.1	0.82	0.001			
	E = 1.55*NL + 1.19	11.0	0.91	0.001			
	E = 1.46*DL + 0.22	10.7	0.92	0.001			
	E = 0.27*NM − 2.20	11.4	0.91	0.001			
Productivity	P = 698*(RNLTOL** − 0.75)	1.29	0.50	0.01			
	P = 727*(RNMTOM** − 0.55)	1.32	0.38	0.02			
Documentation	DOC = 30.4*(L**0.90)	1.41	0.92	0.001	DOC = 49*(L**1.01)	2.68	0.62
	DOC = 38.1*(NL**0.93)	1.52	0.885	0.001			
	DOC = 34.7*(DL**0.93)	1.45	0.91	0.001			
	DOC = 1.54*(M**1.16)	1.45	0.91	0.001			
	DOC = 4.82*(NM**0.99)	1.67	0.83	0.001			
Project duration	D = 4.55*(L**0.26)	1.36	0.55	0.01	D = 4.1*(L**0.36)	1.72	0.41
	D = 1.96*(M**0.33)	1.37	0.54	0.01			
	D = 4.62*(NL**0.28)	1.33	0.61	0.01			
	D = 4.58*(DL**0.28)	1.34	0.59	0.01			
	D = 2.5*(NM**0.30)	1.38	0.55	0.01			
	D = 4.39*(E**0.26)	1.37	0.52	0.01	D = 2.47*(E**0.35)	1.52	0.60
Staff size	S = 0.24*(E**0.73)	1.38	0.89	0.001	S = 0.54*(E**0.6)	1.56	0.79

[a]Standard error of estimate.

[b]Coefficient of determination.

ing a normal distribution of the deviations from the estimation line, we may expect to find about two-thirds of the items (ideally 68.27%) within the band +SE to −SE about the line of estimation, about 95% (ideally 95.45%) within the wider band that includes +2*SE to −2*SE, and practically all (99.73%) within +3*SE to −3*SE. The standard error of estimate is a general or overall measure of the dispersion of all of the Y values around the estimating equation but is often used to indicate the dependability of specific estimates.

The coefficient of correlation expresses the degree of relationship between the two variables. The coefficient of correlation varies from +1 to −1. The sign indicates whether the two variables are directly correlated (positive) or inversely correlated (negative), while the magnitude of the coefficient indicates the degree of association. When there is absolutely no relationship between the variables, $r = 0$. A perfect correlation between the variables is indicated when the magnitude of $r = 1$. The coefficient of determination (r**2) is the amount of variation that has been explained by the line of relationship; 1 − (r**2) is that part of the total variation that has not been explained.

Some of the relationships are illustrated by diagrams (for example, see Figure 1). Each + represents the data from one of the completed projects. The solid line is the estimating equation, or line of regression, computed as described above. The broken lines represent bounds of one standard error of estimate from the estimating equation. The estimating equation, standard error of estimate, and coefficient of determination r^2 are shown in Table 1.

Those relationships also studied by Walston and Felix are illustrated by a diagram comparing their estimating equation with the SEL equation. In Figure 2 each + represents the data from one of the completed SEL projects. The solid line represents the Walston–Felix estimating equation. The two broken lines parallel to the estimating equation represent bounds of one standard error of estimate from the Walston–Felix estimating equation. The other broken line (with finer dash structure) represents the SEL estimating equation.

The derived estimating equations could be used in the following manner. After the project estimates have been computed, those estimates can be checked against the equations that provide an independent estimate based on past experience. Project personnel can then compare these with their own estimates. For example, assume that the size of a delivered software product is estimated by project personnel as 100,000 lines of source code and the effort has been estimated as 200 man-months. However, based on the equation in Figure 1, the estimated total effort for a 100,000-line system should be about 100 man-months. The significant difference between the two estimates does not necessarily imply an error on the part of the project personnel, but it does suggest that the assumptions and estimates leading to the project personnel estimate might be reexamined.

The estimating equations presented here should be considered initial approximations, applicable only to the same environment that the subject projects are from. As data for more projects become available, the estimating equations should be updated and refined.

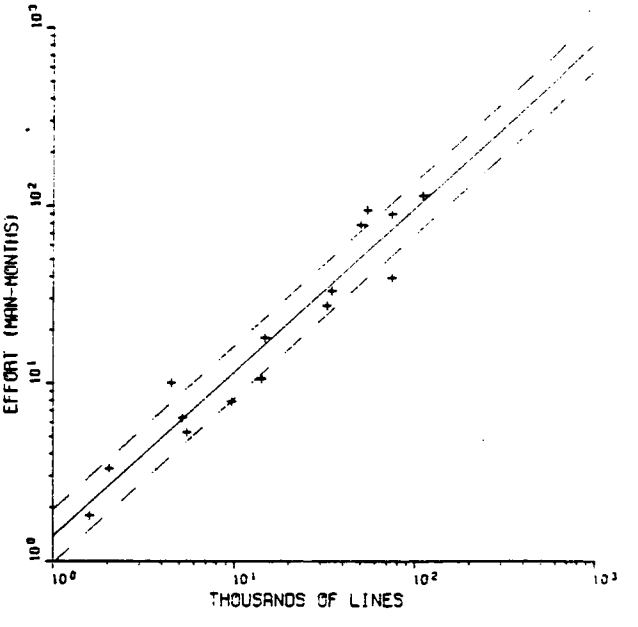
Effort

Effort vs Total Lines. The relationship between delivered source lines of code and total effort is shown in Figure 1. The relationship derived from the data is

$$E = 1.38*(L**0.93). \quad (1)$$

The standard error of estimate can be used to get bounds on the predicting equation. For example, here the standard error of estimate is 1.41, so the coefficient of the exponential term should be multiplied by 1.41 to get an upper-bound equation and divided by 1.41 to get a lower-bound equation. This gives the equations $E = 1.95*(L**0.93)$ and $E = 0.98*(L**0.93)$ as bounds of one standard error of estimate from the estimating equation (1). The coefficient of determination (r**2) is a significantly high (at the 0.001 level) 0.93, indicating (at least for these projects) that there is a high probability of a relationship between total effort and deliv-

Figure 1. Effort vs lines of code: $E = 1.38*(L**0.93)$.



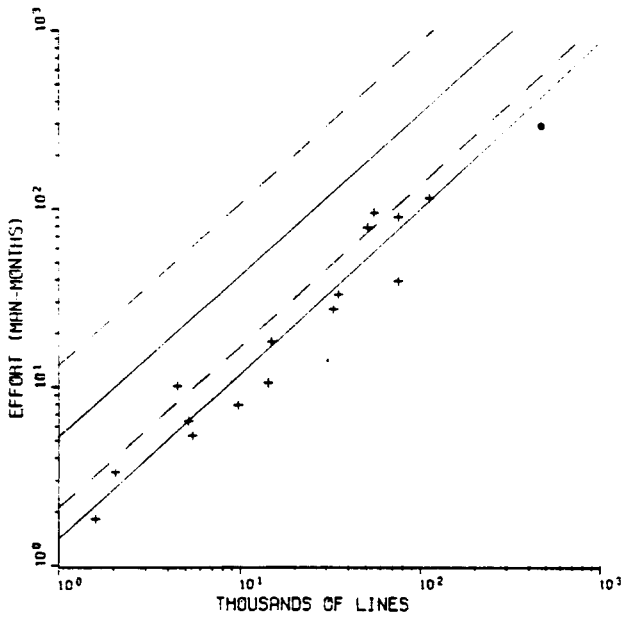


Figure 2. Effort vs lines of code.

ered lines of source code. This relationship is nearly linear.

A linear fit of the data yields

$$E = 1.04*L + 2.04. \quad (2)$$

For a linear fit, the standard error of estimate should be added to the constant term. Thus, the standard error of estimate of 16.1 would give the equations $E = 1.05*L - 14.06$ and $E = 1.05*L + 18.14$ as bounds of one standard error of estimate. However, it is not statistically valid to report a standard error of estimate directly from a least squares linear fit since the points are not uniformly distributed around the prediction line. An additive standard error would be unreasonable, since it would be too small for large projects and too large for small projects.

Walston and Felix also found a nearly linear relationship between total effort and product size:

$$E = 5.2*(L**0.91) \quad (3)$$

with a standard error of estimate of 2.51. This places the equation derived from the SEL data somewhat below one standard error of estimate of the Walston–Felix equation (see Figure 2). Equation (1) seems to indicate that less effort is required than predicted by (3) to develop the same amount of product. A possible explanation is that the projects studied by Walston and Felix were very diversified; that is, there were many different types of programs [5]. In the SEL environment, however, the programs are almost all of the same general type, and the project personnel have experience developing this type of software, implying there may be less design effort required. In the Walston–Felix study, however, many of the projects were of the large, complex, one-time custom program type where the problems and their solutions are not well understood.

Effort vs New Lines and Developed Lines. Some programming projects reuse code from previous projects in an attempt to reduce the total effort required to produce a system. The relationship between total effort and thousands of new delivered source code,

$$E = 1.58*(NL**0.99), \quad (4)$$

is also nearly linear and has a high coefficient of determination.

A linear fit of the data gives

$$E = 1.55*NL + 1.19. \quad (5)$$

Substituting developed lines for new lines, the equations become

$$E = 1.48*(DL**0.98), \quad (6)$$
$$E = 1.46*DL + 0.22. \quad (7)$$

(See Table 1 for standard error and coefficient of correlation values.)

The relationships between total effort and total new and developed lines of source code have high coefficients of determination, indicating that they could be used to predict the total effort if the number of lines of source code (either total or new) could be determined beforehand.

Effort vs Modules. Another measure of program size is the number of modules in the product. Total effort and the number of modules in the delivered product are related as shown in Figure 3:

$$E = 0.65*(M**1.19). \quad (8)$$

The relationship is not as linear as that between total effort and delivered lines of source code, but the coefficient of determination indicates there is a high correlation between the total effort and the number of modules in the delivered product. A similar relationship exists between total effort and the number of new modules:

$$E = 0.183*(NM**1.05). \quad (9)$$

A new module is defined as a completely new module or one used from a previous project and having more than 20% of the module changed. A linear fit of the data gives

$$E = 0.27*NM - 2.20. \quad (10)$$

These equations may be more useful as estimating

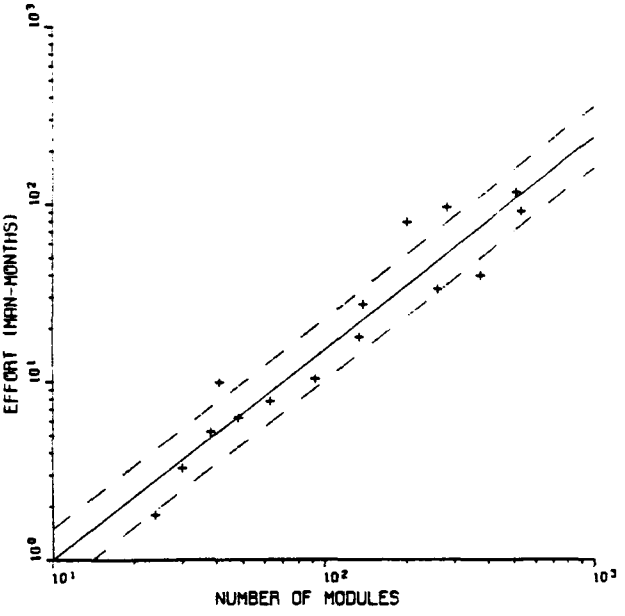


Figure 3. Effort vs number of modules: $E = 0.65*(M**1.19)$.

aids than (1) and (3) since it is more likely that the number of modules (or a good approximation) is known early in the project life cycle, particularly after the preliminary design phase.

In all the above relationships between effort and size there appears to be a linear relationship independent of the particular size measure. This means that productivity remains relatively constant as the size of the project changes. This may seem surprising, but it does support the IBM Federal Systems result.

Productivity

Productivity is one of the most important factors in all software estimating processes. Here productivity is defined as the ratio of delivered source lines of code to the total effort (in man months) required to produce the product. For this environment, productivity, calculated in terms of delivered lines L, new lines (NL), and developed lines (DL), is in the range of 600–700 lines of code per man-month. It must be remembered, however, that this productivity figure includes the design, code, and testing phases only.

Productivity plotted against the ratio of new lines of source code to total delivered lines of source code produces (see Figure 4)

$$P = 698*(\text{RNLTOL}** - 0.75). \quad (11)$$

New code is defined as before. The relationship between the two variables suggests that productivity is lowest when there is no reused code. As the percentage of reused code increases, the expected overall productivity increases. This reinforces the intuitive idea that the reuse of a code should be less expensive than creating the code from scratch. The coefficient of determination (0.50) is significant at the 0.01 level.

The Walston–Felix definition of reused code is related more to size change above the original rather than code added, which is significantly different from that used in this paper, so a comparison of the two results would be meaningless.

The relationship between productivity and the ratio of new modules to total modules is

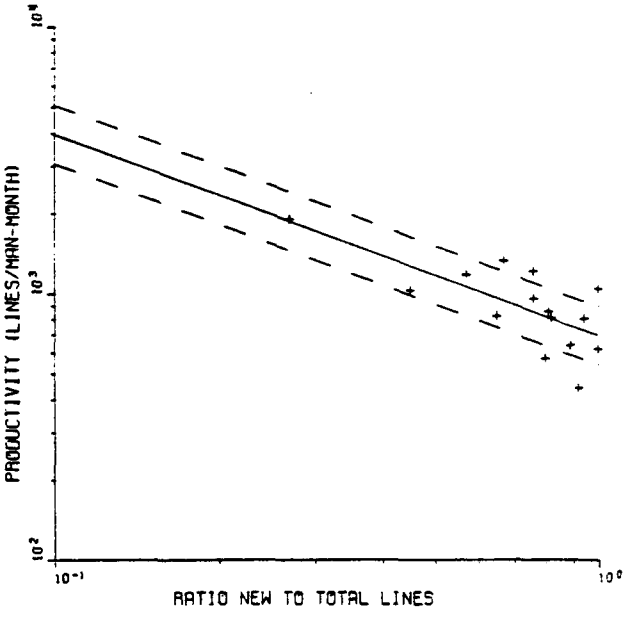
$$P = 727*(\text{RNMTTOM}** - 0.55). \quad (12)$$

New modules are defined as before. This relationship exhibits the same behavior as (11). The coefficient of determination is significant at the 0.02 level.

Documentation

Documentation is an important part of any software project, and the costs of producing documentation are a factor in the software estimating process. The size of documentation is measured in pages. Here, documentation is defined as the program design (handwritten), test plan, user guide, system description, and module description. The module description contains a one-page description of each module. Figures 5 and 6 show the number of pages of documentation vs thousands of

Figure 4. Productivity vs RNTOL: $P = 698*(\text{RNTOL}** -0.75)$.



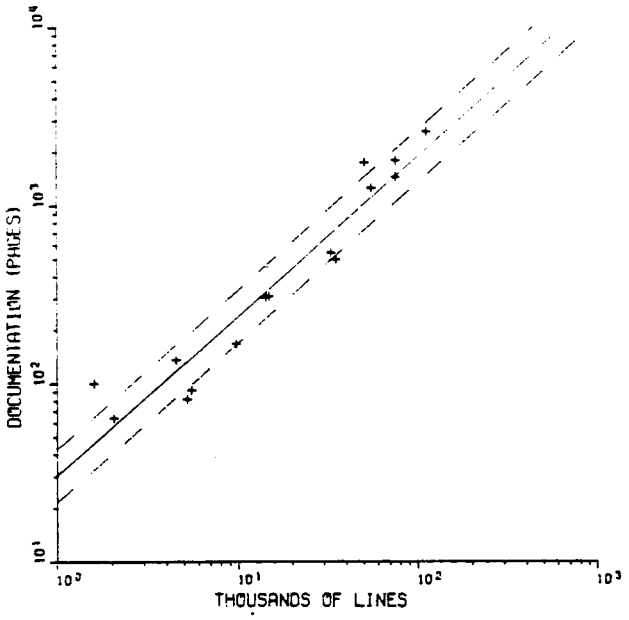


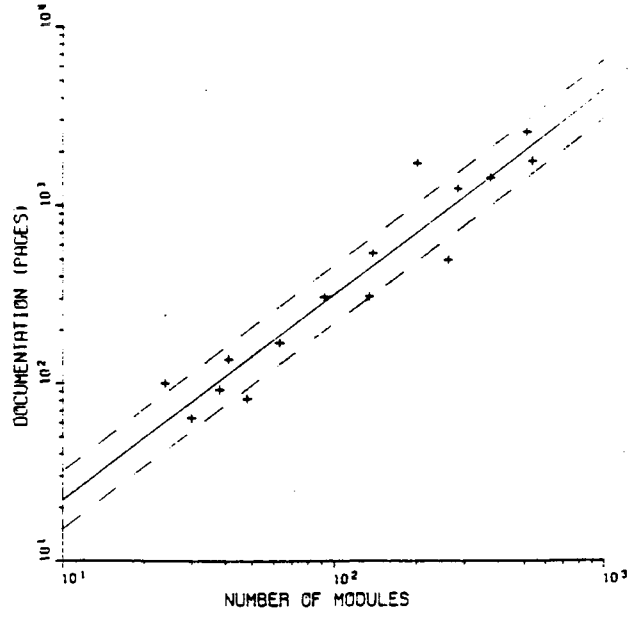
Figure 5. Documentation vs lines of code: DOC = 30.4*(*L***0.90).

delivered lines of source code and number of modules, respectively. The correlation equations are

DOC = 30.4*(*L***0.90), (13)
DOC = 1.54*(*M***1.12). (14)

Both relationships are roughly linear and have coefficients of determination significant at the 0.001 level.

Figure 6. Documentation vs number of modules: DOC = 1.54*(*M***1.12).



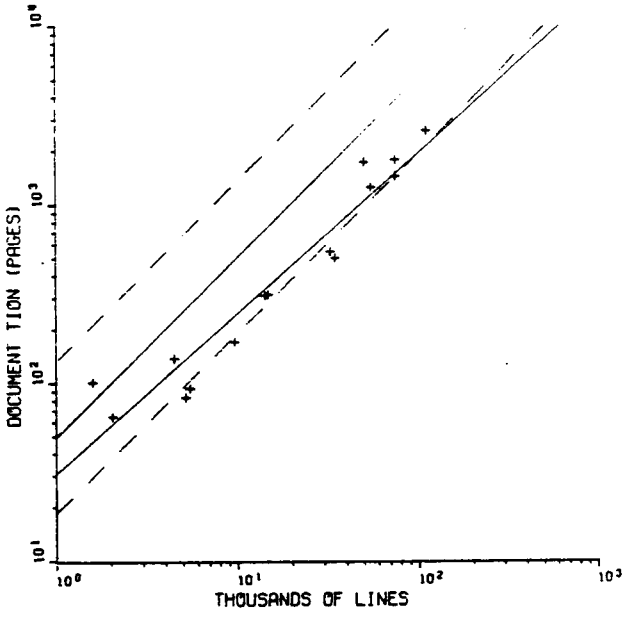


Figure 7. Documentation vs lines of code.

Walston and Felix also found that a nearly linear relationship exists between the number of pages of documentation and the number of thousands of delivered lines of source code:

DOC = 49*(*L***1.01). (15)

Figure 7 shows a comparison of (13) and (15). The SEL equation lies about one standard error of estimate below the Walston–Felix equation. Part of the difference may be explained by the fact that Walston and Felix included in their definition of documentation such items as flowcharts and source program listings, which are not included in the SEL documentation page counts.

Documentation as a function of each remaining size measure, new lines, developed lines, and new modules is

DOC = 38.1*(NL**0.93), (16)
DOC = 34.7*(DL**0.93), (17)
DOC = 4.82*(NM**0.99), (18)

respectively. Again, notice that these relationships are approximately linear. The coefficients of determination are significant at the 0.001 level.

Duration

The problem of determining the duration of a software project is difficult and important. The relationship between project duration (in months) and number of thousands of lines of source code is shown in Figure 8,

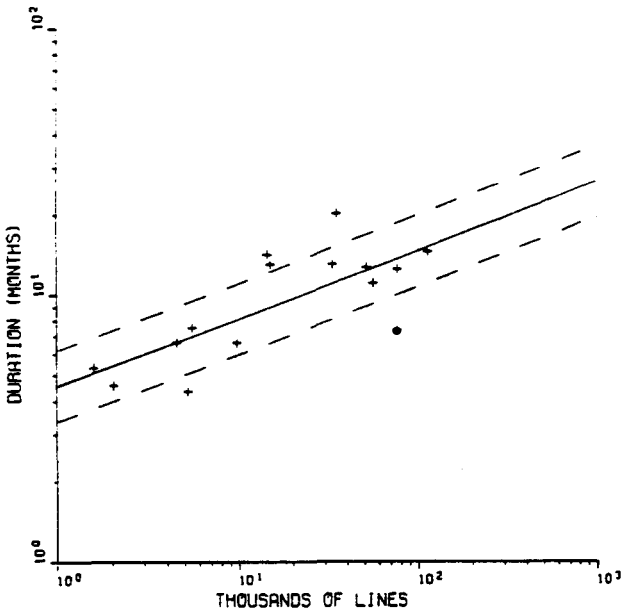


Figure 8. Duration vs lines of code: $D = 4.55*(L**0.26)$.

and the relationship between duration and the number of modules is shown in Figure 9. The equations for these relationships are

$$D = 4.55*(L**0.26), \quad (19)$$
$$D = 1.96*(M**0.33), \quad (20)$$

respectively. Walston and Felix found a nearly cubic relationship between project duration and delivered code:

$$D = 4.1*(L**0.365). \quad (21)$$

This relationship is quite similar to that found for the SEL data (see Figure 10).

Reusing code or modules may have an effect on project duration. The relationships between project duration and new lines of code in thousands, developed lines in thousands, and new modules are

$$D = 4.62*(NL**0.28), \quad (22)$$
$$D = 4.58*(DL**0.28), \quad (23)$$
$$D = 2.5*(NM**0.30), \quad (24)$$

respectively. These relationships are very close to (19) and (20). As one might expect, calendar time increases at about one-third the rate of size. This is owing to the fact that calendar time on larger projects is a major constraint and more people are required to meet the calendar deadlines.

Project duration as a function of total effort is shown in Figure 11. The regression equation is

$$D = 4.39*(E**0.26). \quad (25)$$

Walston and Felix also found that a cubic relationship exists between project duration and total effort:

$$D = 2.47*(E**0.35). \quad (26)$$

Equations (25) and (26) are very similar. A comparison of the two estimating equations is shown in Figure 12. The SEL equation lies about one standard error of es-

Figure 9. Duration vs number of modules: $D = 1.96*(M**0.33)$.

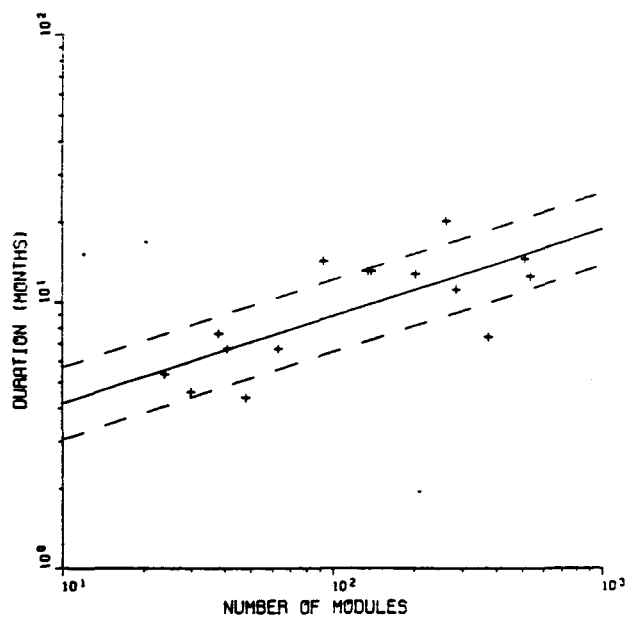
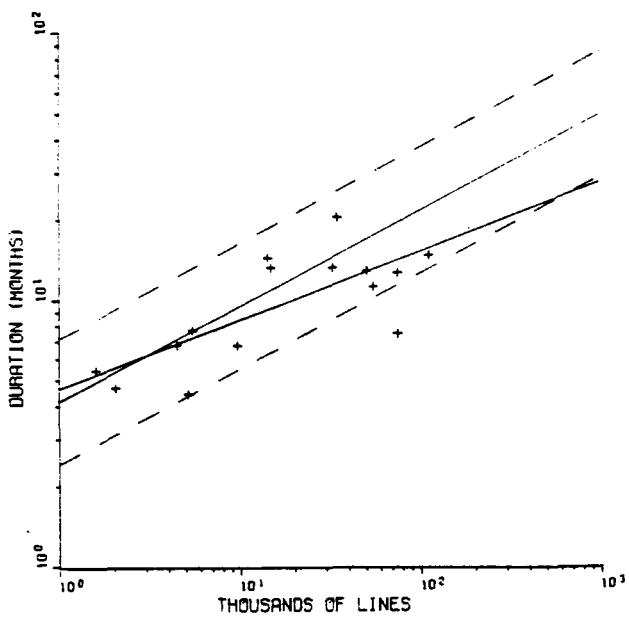


Figure 10. Duration vs lines of code.



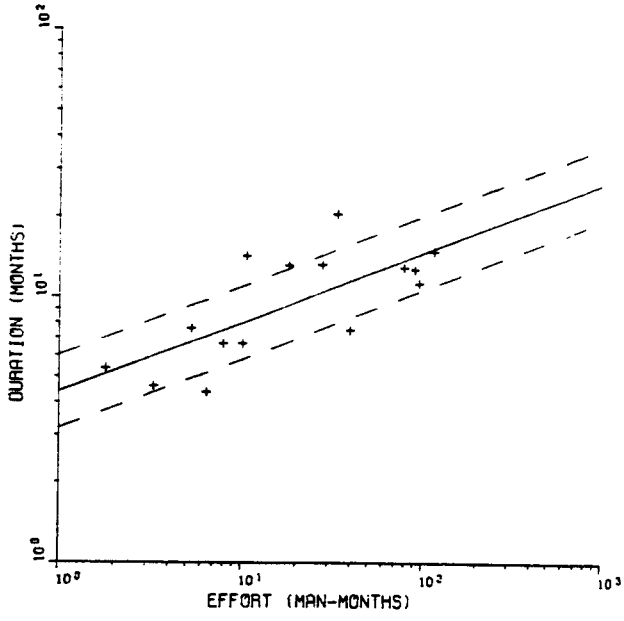


Figure 11. Duration vs effort: $D = 4.39*(E**0.26)$.

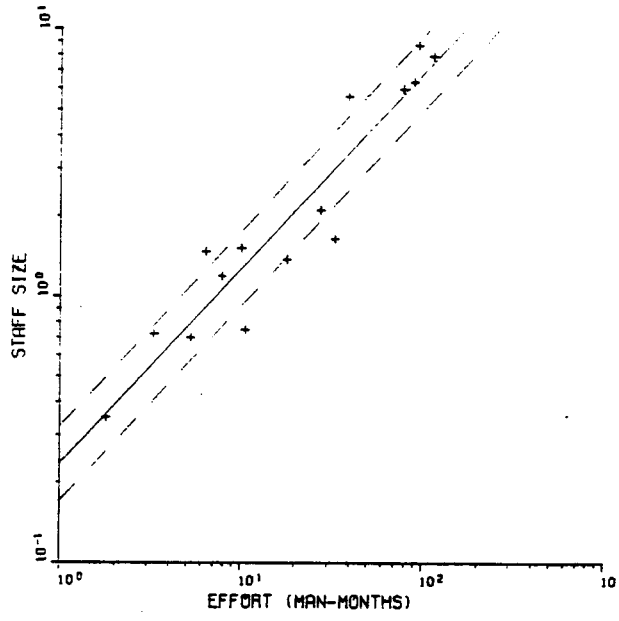


Figure 13. Staff size vs effort: $S = 0.24*(E**0.73)$.

timate above the Walston–Felix equation. More will be said about this relationship in the next section.

Staff Size

The staff size used for the development of a software product depends on several factors, including the development time allowed for the project, the amount and difficulty of the code to be produced, and the manpower loading rates that can be achieved [10]. The equation relating average staff size (total man months of effort divided by the project duration in months) and total effort (Figure 13) is

$$S = 0.24*(E**0.73). \quad (27)$$

The coefficient of determination indicates a good rela-

Figure 12. Duration vs effort.

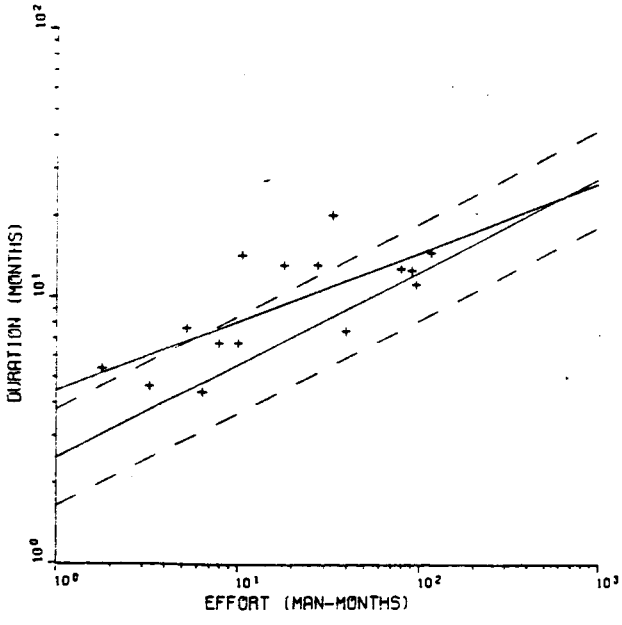
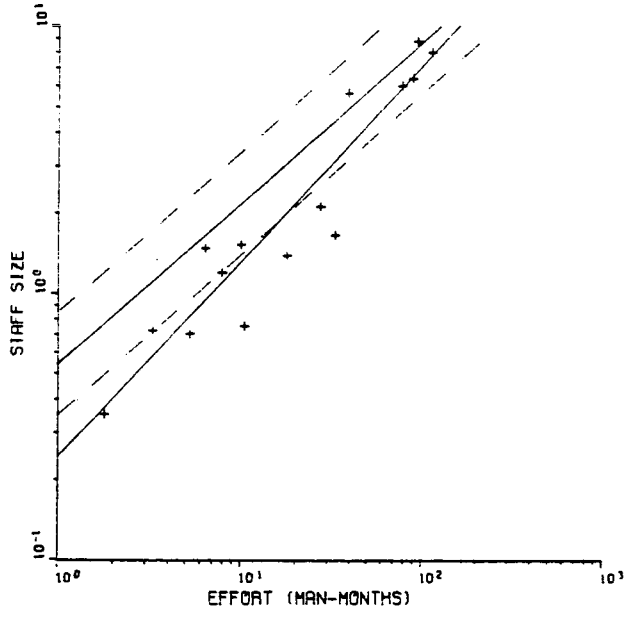


Figure 14. Staff size vs effort.



tionship between these two variables. The Walston–Felix equation for this relationship is

$$S = 0.54*(E**0.6). \quad (28)$$

Again, this equation is very much like (27). Figure 14 shows the SEL estimating equation and (28) together. The SEL equation lies about one standard error of estimate below the Walston–Felix equation. The Walston–Felix equation shows consistently higher manning levels than the SEL equation, but the Walston–Felix equation relating project duration and total effort shows consistently shorter project durations for the same amount of expended effort (Figure 12). Thus, the shorter project durations in the Walston–Felix study seem to have been gained by higher staffing levels.

CONCLUSIONS

The authors believe they have been able to help validate the basic relationships reported by Walston and Felix [5] in their original study. Clearly, the equations' coefficients are different for different environments, as one would expect, but there is a consistency in the way the SEL equations relate to the Walston–Felix equations.

On the other hand, the SEL data could also be used to support the linear relationship between effort and lines of code described by Jeffery and Lawrence [4]. Their data deal with business applications, predominantly in COBOL, ranging in size from 100 to 4500 lines of code. Their effort includes detailed design, coding, and testing. The SEL data lies between the Jeffery–Lawrence data and the Walston–Felix data with respect to size.

Whether the relationship between effort and lines of code is modeled by a linear equation ($E = a*L + b$) or an exponential equation $E = a*(L**b)$, it is closer to linear than one might expect. For example, it has been hypothesized [11] that the relationship is more exponential and of the form

$$E = a*(L**1.5).$$

The basis of this hypothesis is that as the problem gets larger it becomes more difficult to develop the solution, and so the effort per line of code should increase. Implicit in this assumption is that lines of code is a measure of function complexity and that the relationship between the two is linear.

However, it is possible that this last assumption is false. As the problem increases in size and complexity, the size of the code may increase at an even greater rate. This increased rate is due to subfunction duplication and the looseness of the code. For example, as the problem increases in size and more people are involved in the development, it becomes more difficult to recognize duplicate function. Much of this duplicate function may be simple routines that each programmer redevelops for himself. As the complexity of the function increases, as it may very well do with size, there may be a looseness of code, a tendency to write a longer, simpler algorithm to keep the system simple. There are limits to the amount of complexity an individual can handle. It is also often true that there tends to be more overdesign of subprograms. The insecurity caused by the pure size forces the programmer to overdesign for safety, which results in more code per function. All of this extra code creates a larger system whose relation to the problem grows exponentially with respect to the size of the problem. Thus, the equation

$$E = a*(\text{function}**b),$$

where b is about 1.5, may be true, but when compared with size measured as lines of code b is closer to 1. Unfortunately, we are unable to measure function and complexity accurately enough to verify this hypothesis.

Some comments on the basic relationships seem worth making:

a. Based on the SEL data, it appears that developed lines and new lines are a better estimate of effort than total lines. This is intuitively satisfying since it is closer to the notion of expended effort.
b. Even though the measure of productivity is rather primitive, there is a tendency to believe from our data that reusing code is cost effective. Because of different ways of counting reused code, we were unable to compare our data with that of Walston and Felix.
c. The use of modules as a measure of effort works about as well for the SEL environment as various measures of lines of code. Since in many cases it is easier to predict the number of modules than lines of code, this provides a viable approach to prediction.
d. Productivity in environments where the design is better understood may increase by a factor of 3 or 4.
e. On large projects, calendar time is a major factor. It increases with the cube root of effort.
f. The relationships between documentation and product size, between duration and effort or size, and between staff size and effort are reasonably supportive of the Walston–Felix relationships.

This approach to estimation is empirically based, and the data are highly dependent on the local environment. It is an indicator of how we currently do business and defines the common aspects of the developmental environment. As new projects are added to the data base, the equations will change and the base relationships will change as the way we do business changes.

The differences between the actual data and the pre-

dicted values of the equations can be explained by variations in the environmental factors for the different projects within the SEL, including methodology and constraints. We can think of the basic lines of code and effort as capturing the essential SEL environment and the individual projects as requiring modification due to specific variations within the project environment. This approach was used by Walston and Felix in their productivity index and by Boehm ([12]) in his COCOMO model. We are currently investigating this approach by developing a metamodel that will be adapted to the local organizational and project environment by isolating local SEL environmental factors.

ACKNOWLEDGMENTS

The authors are grateful to Frank McGarry of NASA/Goddard Space Flight Center, Jerry Page and Victor Church of Computer Sciences Corporation, Claude Walston and Charles Felix of IBM Federal Systems Division, and John Bailey of the University of Maryland for their comments on drafts of this paper and for supplying data needed for this paper.

REFERENCES

1. V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, W. F. Truszkowski, and D. L. Weiss, *The Software Engineering Laboratory*, Tech. Rep. TR-535, Department of Computer Science, University of Maryland, May 1977.
2. V. R. Basili and M. V. Zelkowitz, Analyzing Medium Scale Software Development, *Proceedings of the Third International Conference on Software Engineering*, Atlanta, Georgia, May 1978, pp. 116–132.
3. K. Freburger and V. R. Basili, *The Software Engineering Laboratory: Relationship Equations*, Tech. Rep. TR-764, Department of Computer Science, University of Maryland, May 1979.
4. D. R. Jeffery and M. J. Lawrence, *An Inter-organizational Comparison of Programming Productivity*, Department of Information Systems, University of New South Wales, 1979.
5. C. E. Walston and C. P. Felix, A method of Programming Measurement and Estimation, *IBM Syst. J.* 16, 1 (1977).
6. E. Chrysler, Some Basic Determinants of Computer Programming Productivity, *Commun. ACM* 21, 6 (1978).
7. J. R. Johnson, A Working Measure of Productivity, *Datamation* 23, 2 (1977).
8. V. LaBolle, *Development of Equations for Estimating the Costs of Computer Program Production*, System Development Corporation, Santa Monica, California, 1966.
9. E. A. Nelson, *Management Handbook for the Estimation of Computer Programming Costs*, System Development Corporation, Santa Monica, California, 1967.
10. L. H. Putnam, A General Empirical Solution to the Macro Software Sizing and Estimating Problem, *IEEE Trans. Software Eng.* SE-4, 345–361 (1978).
11. F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Massachusetts, 1975.
12. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

EVALUATING AND COMPARING SOFTWARE METRICS IN THE SOFTWARE ENGINEERING LABORATORY*

Victor R. Basili and Tsai-Yun Phillips
Department of Computer Science
University of Maryland
College Park, MD 20742

*Research supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

EVALUATING AND COMPARING SOFTWARE METRICS IN THE SOFTWARE ENGINEERING LABORATORY

I. Introduction

There has appeared in the literature a great number of metrics that attempt to measure the effort or complexity in developing and understanding software [1]. There have also been several attempts to independently validate these measures on data from different organizations gathered by different people [2]. These metrics have many purposes. They can be used to evaluate the software development process or the software product. They can be used to estimate the cost and quality of the product. They can also be used during development and evolution of the software to monitor the stability and quality of the product.

Among the most popular metrics have been the software science metrics of Halstead, and the cyclomatic complexity metric of McCabe. One question is whether these metrics actually measure such things as effort and complexity. One measure of effort may be the time required to produce a product. One measure of complexity might be the number of errors made during the development of a product. A second question is how these metrics compare with standard size measures, such as the number of source lines or the number of executable statements, i.e., do they do a better job of predicting the effort or the number of errors? Lastly, how do these metrics relate to each other?

One simple way of checking the relationship between errors or effort and the various metrics is to examine the plots of the variables against one another and correlations between the various variables. This provides us with a first look at attempting to shed some light on the questions posed and the relationships that may hold.

One of the goals of the Software Engineering Laboratory [3] has been to provide an experimental data base to be used for examining such relationships and providing insights into attempting to answer such questions. The Software Engineering Laboratory is a joint venture betwen the University of Maryland, NASA/Goddard Space Flight Center, and Computer Sciences Corporation.

The software being analyzed is ground support software for satellites. The systems in this paper consist of 50,000 to 110,000 lines of source code. The source code is predominantly FORTRAN. Anywhere from 10 to 60 percent of the code is reused from previous systems. There are between 200 and 500 modules in each system where a module is defined as a FORTRAN subroutine. The average staff size ranges from 5 to 8 people, including support personnel.

II. The Data

Data is collected in the Software Engineering Laboratory that deals with many aspects of the development process and product. Among the data collected is the effort to design, code and test the various components of the systems as well as the errors committed during development. The data collected is analyzed to provide insights into software development and to study the effect of various factors on the process and product.

One standard problem in data of this kind is its validity. Unlike the typical controlled experiments where the projects tend to be smaller and the data collection process dominates the development process, the major effort here is the software development process, and the data collectors must effect minimal interference to the developers.

This creates potential problems with the validity of the data. For example, suppose we are interested in the effort expended on a particular module and one programmer forgets to turn in his weekly effort report. This can cause erroneous data for all modules the programmer may have worked on that week. Another problem is how does a programmer report time on the

integration testing of three modules? Does he charge the time to the parent module of all three, even though that module may be a small driver module? Clearly that is easier for him to do than to divide the time between all three modules he has worked on.

How does one count errors? An error that is limited to one module is easy to credit. But what about an error that required the analysis of ten modules to determine that it effects changes in three modules? Does one associate the error with all ten modules or the three modules? Does one associate one third of an error with each of the three modules or a full error with all three? It is clear that the larger the system the more complicated the association. All this assumes that all the errors were reported. It is common for programmers not to report clerical errors because the time to fill out the error report form might take longer than the time to fix the error.

In a commercial program development environment, the errors are not seeded so they are not known to the analysis beforehand. The programmers are not watched with respect to the time they put in and report; the full development process may take a year. A class of problems not expected in the controlled development environment is common here and can create problems with obtaining valid results.

The data discussed in this paper is extracted from several sources. First, there is effort data which is taken from a form called the Components Status Report. This report is filled out each week by the programmers on the project. They report the time they spend on each component in the system broken down into the basic phases of design, code and test, as well as any other time they spend on work related to the project, e.g., documentation, meetings, etc.

A component is defined as any named object in the system. A component could be a FORTRAN subroutine, a COMMON block or a set of subroutines that makes up a subsystem. The effort data analyzed in this paper is extracted from the Component Status Reports.

Another form, filled out weekly by the project management, is the Resource Report Form. This form represents accounting data and records all time charged to the project for the various project personnel. It is not broken down by activity. This data is used in section IV of this paper to validate the effort data on the components.

The varous metrics computed on the source data are calculated automatically by a program called SAP [4] which was developed specifically for the Software Engineering Laboratory by Computer Sciences Corporation. Data collected by the SAP program consists of various software science metrics, such as the Halstead E metric used here, the number of decisions (similar to the McCabe cyclomatic complexity metric), the number of source statements, the number of executable statements and the number of call statements. These metrics are computed at the component level. The number of source lines consists of the total number of lines of source text, including comments and data statements. The number of executable statements consists of only the executable FORTRAN statements, excluding comments and data statements such as COMMON declarations. Typically, the number of executable statements is about 50 percent to 60 percent of the total number of source lines.

The error count discussed here is collected from a form called the Change Report Form which is filled out each time a change is made to the system. These reports are normally not filled out until testing has begun. The error count consists of only those changes which have been classified as errors. Nonerror changes are not discussed in this paper.

III. A First Pass

We began by examining four projects which we shall call A, I, P and S. For each of these projects we considered the aspects of the development separately and in combinations. These phases are the design, coding, and testing

phases. In considering all available components, A had 111 components, I had 55 components, P had 229 components, and S had 118 components for which we had some effort data and a software science E measure. It turned out that the union of coding and testing, as well as total effort, gave us the best results:

Project	Design	Code	Test	Design & Code	Design & Test	Code & Test	Total
A	.4563	.4700	.4212	.4775	.5444	.6380	.6599
I	-.0503	.0322	.0094	.0931	.0942	.0977	.0500
P	.3817	.4316	.3946	.4301	.4296	.4296	.4660
S	.3658	.3957	.4015	.4157	.4688	.4688	.5459

The lowest correlations between effort and the E metric were in project I. As it turns out, project I had the most reused code from previous projects. That is, modules from previous projects were taken wholly or slightly modified for use in system I. Since this factor was obviously affecting the relationship, we classified all the modules studied as either newly developed, modified, or old. We then recalculated the relationship between effort and the E metric using only newly developed components. The results are given below for total effort only.

Project	# of Components	Total
A	101	.6774
I	31	.4162
P	178	.6230
S	106	.4580

The correlations here are higher, as expected, because of the better data.

We were interested in whether other measures, such as lines of source code and executable statements, provided better relationships as well as the

relationship between the E metric and these other measures. We were also interested in whether other factors affected the correlations. For example, what effect would a division of the modules by such factors as size, complexity and testing level have?

First, a study of all 416 components across the four systems yielded the following correlations:

	E	Source Lines	Executable Statements
E		.7497	.8031
Actual Effort	.6384	.5795	.4949

Next, division of the components by the amount of time spent in development effort showed better correlations for those projects in which more time was spent. The division by the number of lines, however, did not show a clear trend. This provided us with the idea that some of the effort data might be missing at the component level and, therefore, we should eliminate those components for which the effort data was not good enough. The results of this validation are reported in the next section.

The separation of components by complexity was based upon an evaluation of the complexity of the particular component by the programmer. Components were rated as hard, moderate or easy. In general, higher correlations between effort and all other variables grew as the subjective complexity rating grew.

The results of separating the components by various subsystems that were common across the projects, as well as by various testing levels (such as tree chart subsystem levels), seemed inconclusive. These variations will be examined again in light of the data validation discussed in the next section.

IV. A Second Pass

Because the correlations between effort and the various size metrics were better for those components with greater effort, we became concerned that the results might be due to poor reporting of effort data. To check this, we proposed a validation check on the data, providing each component with a validity rating. For each programmer on a project, we examined both the total time reported on the Component Status Report, as well as the total time charged to the project. We then placed components into categories depending upon the percent of time reported by the programmer on the Component Status Report compared to the percent of time charged to the project, and gave the components an accuracy rating. For example, if all the programmers working on component X reported at least 90 percent of their total resource time on the Component Status Report, then X is in the $\geq$ 90 percent category.

Besides examining the E metric, the source lines and executable statement counts, we also analyzed the cyclomatic complexity metric and the number of calls contained within a component. The correlation between actual effort and these complexity metrics is given in table 1(a) and (b) for those projects with greater than 90 percent accuracy and greater than 84 percent accuracy. Figures 1, 2, 3, and 4 provide plots of the data points at the 84 percent and 90 percent accuracy levels for source lines and the E metric with effort.

The correlations between the various factors appear to be better on the validity rated data than on the full data and appears to do better at the 90 percent validity rated level than at the 84 percent validity rated data. For this reason, we believe that the validity rated data is more reliable than the earlier data.

Since complexity is also meant to measure the number of errors associated with the development of a project, we compared the various complexity measures,

Pearson Correlation Coefficients

≥ 90% Reported Programmers

	Effort	Error	Halstead	XQT	Source	McCabe -1	Calls
Effort	1.0000*	.6346*	.6612*	.7974*	.7583*	.7399*	.6033*
Error	.6346*	1.0000*	.5432*	.5837*	.5576*	.5592*	.4861*
Halstead	.6612*	.5432*	1.0000*	.9160*	.8706*	.8906*	.8818*
XQT	.7974*	.5837*	.9160*	1.0000*	.9513*	.9777*	.8258*
Source	.7583*	.5576*	.8706*	.9513*	1.0000*	.9519*	.8726*
McCabe -1	.7399*	.5592*	.8906*	.9777*	.9519*	1.0000*	.8110*
Calls	.6033*	.4861*	.8818*	.8258*	.8727*	.8110*	1.0000*

cases = 37 (data points)

* - significance ≅ .001

Table 1(a)

- -

Pearson Correlation Coefficients

Across Projects (≥ 84% Reported Programmers)

	Effort	Error	Halstead	XQT	Source	McCabe -1	Calls
Effort	1.0000*	.6227*	.6719*	.5094*	.6025*	.3261*	.6666*
Error	.6227*	1.0000*	.5028*	.4289*	.4891*	.3045*	.6431*
Halstead	.6719*	.5028*	1.0000*	.8301*	.7565*	.6540*	.8044*
XQT	.5094*	.4289*	.8301*	1.0000*	.8061*	.9116*	.7703*
Source	.6025*	.4891*	.7565*	.8061*	1.0000*	.6533*	.7759*
McCabe -1	.3261*	.3045*	.6540*	.9116*	.6533*	1.0000*	.5990*
Calls	.6666*	.6431*	.8044*	.7703*	.7759*	.5990*	1.0000*

cases = 116 (data points)

* - significance ≅ .001

Table 1(b)

including the total effort required for development with the number of errors. An error was associated with a component if it was isolated to that component or the component was one of several involved in the error. Table 1 also gives the correlations between the error count and the various complexity metrics. Figures 5, 6 and 7 provide plots of the data points at the 84 percent accuracy level for actual effort, source lines and Halstead's E metric compared with the error count.

Another question is whether we can predict or account for the actual effort or error count using the metrics discussed so far. In an attempt to study this problem we applied a forward multiple regression analysis using the other metrics to account for effort. Using the data for effort at the 84 percent validity level, we came up with the following results. The analysis brought in the variables in the following order: executable statements (XQT), number of errors (ERR), E metric (E), cyclomatic complexity (CC) and source lines. The number of calls was never included. Table 2 shows the amount of variation explained (R^2) as each new factor is included in the equation. Based on a .05 level of significance in using the last variable included, the regression equation generated was

Effort = 19.9 * XQT + 107.5 * ERR - 1.2 * E - 24.7 CC + 250.5

Dependent Variable . . . Effort	
variables	R^2
Executable Statements	.6358
Error Count	.6792
E	.7110
Cyclomatic Complexity	.74571
Source Lines	.74966

Table 2

There has been some work done in isolating the individual programmers. There is some evidence that a better correlation exists between the effort or error count of an individual programmer and a particular complexity metric. Some work will also be done in examining specific error classes and complexity metrics.

Further validation of the data needs to be done in examining some of the outlying points. For example, a point with a high number of source lines but low effort rating might be a COMMON block and therfore eliminated from the study of control flow components.

V. Conclusion

There is hope in using commercially-obtained data rather than experimentally-obtained data to validate complexity metrics. It is possible to systematically clean up the data and study the relationships between effort, error counts and the various complexity metrics. We have shown that some relationships and accountability do exist between various complexity metrics, effort and error counts. The results tend to get better as the data used appears to be more reliable.

References

[1] Halstead, M., Elements of Software Science, Elsevier North-Holland, New York, 1977

McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, 1976, 2, 308-320

Gaffney, John and Heller, Gilbert L., "Macro Variable Software Models for Application to Improved Software Development Management," Proceedings of Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, IEEE Computer Society

Chen, E. T., "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, May 1978, Vol. SE-4, No. 3, pp. 187-194

[2] Curtis, Sheppard, & Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings of the Fourth International Conference on Software Engineering, 1979, pp. 356-360

Feuer and Fowlkes, "Some Results from an Empirical Study of Computer Software," Proceedings of the Fourth International Conference on Software Engineering, 1979, pp. 351-355

Basili, V., "Tutorial on Models and Metrics for Software Management and Engineering," IEEE Computer Society, IEEE Catalog No. EHO-167-7, 1980

[3] Basili and Zelkowitz, "Analyzing Medium Scale Software Developments," Third International Conference on Software Engineering, Atlanta, Georgia, May 1978

[4] O'Neil, E., "The Static Source Code Analyzer's Users Guide," CSC TM-78/6045, 1978

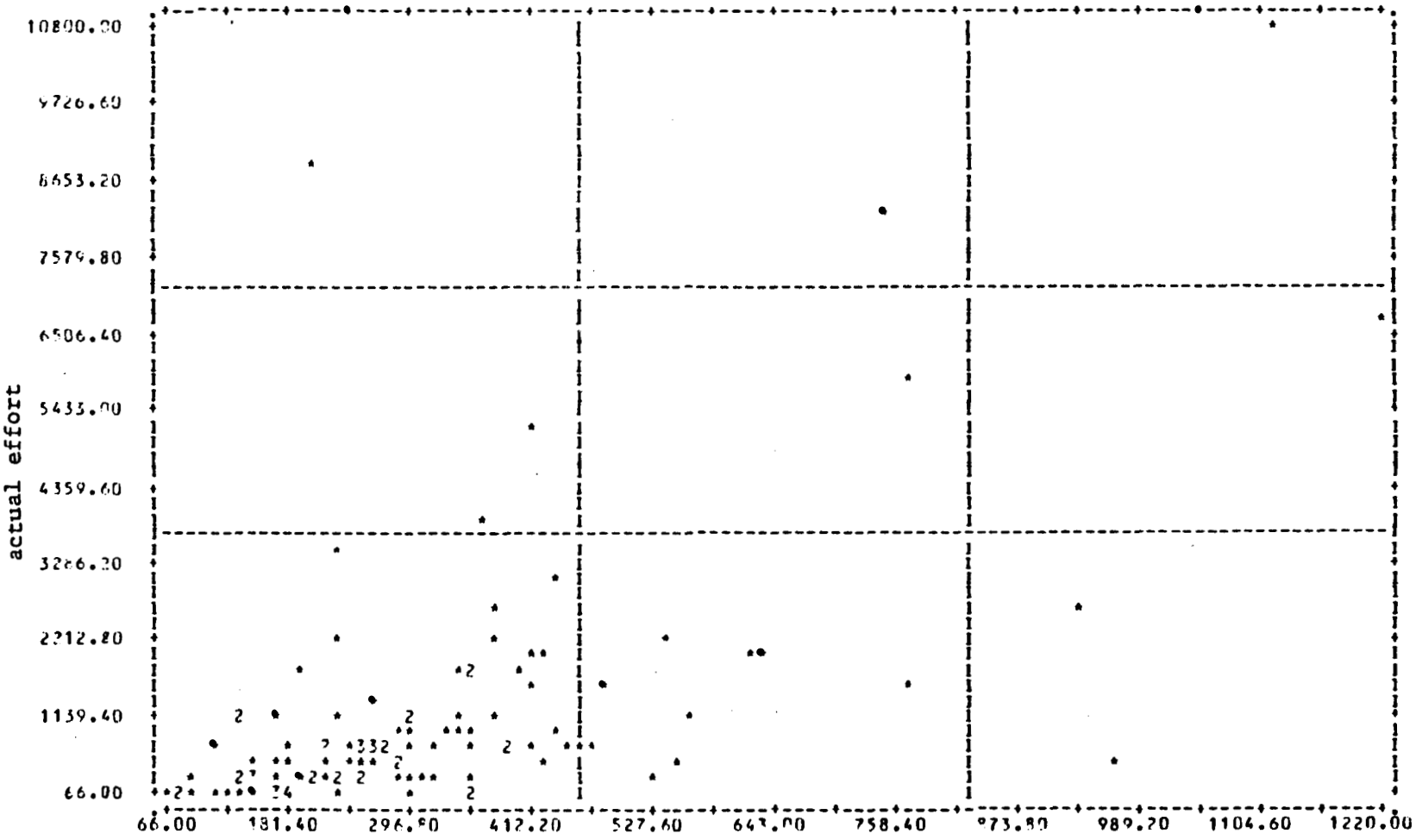


Figure 1

84% accuracy level
Correlation$_2$ (R): .6025
R square (R^2): .3630
Significance: .00001

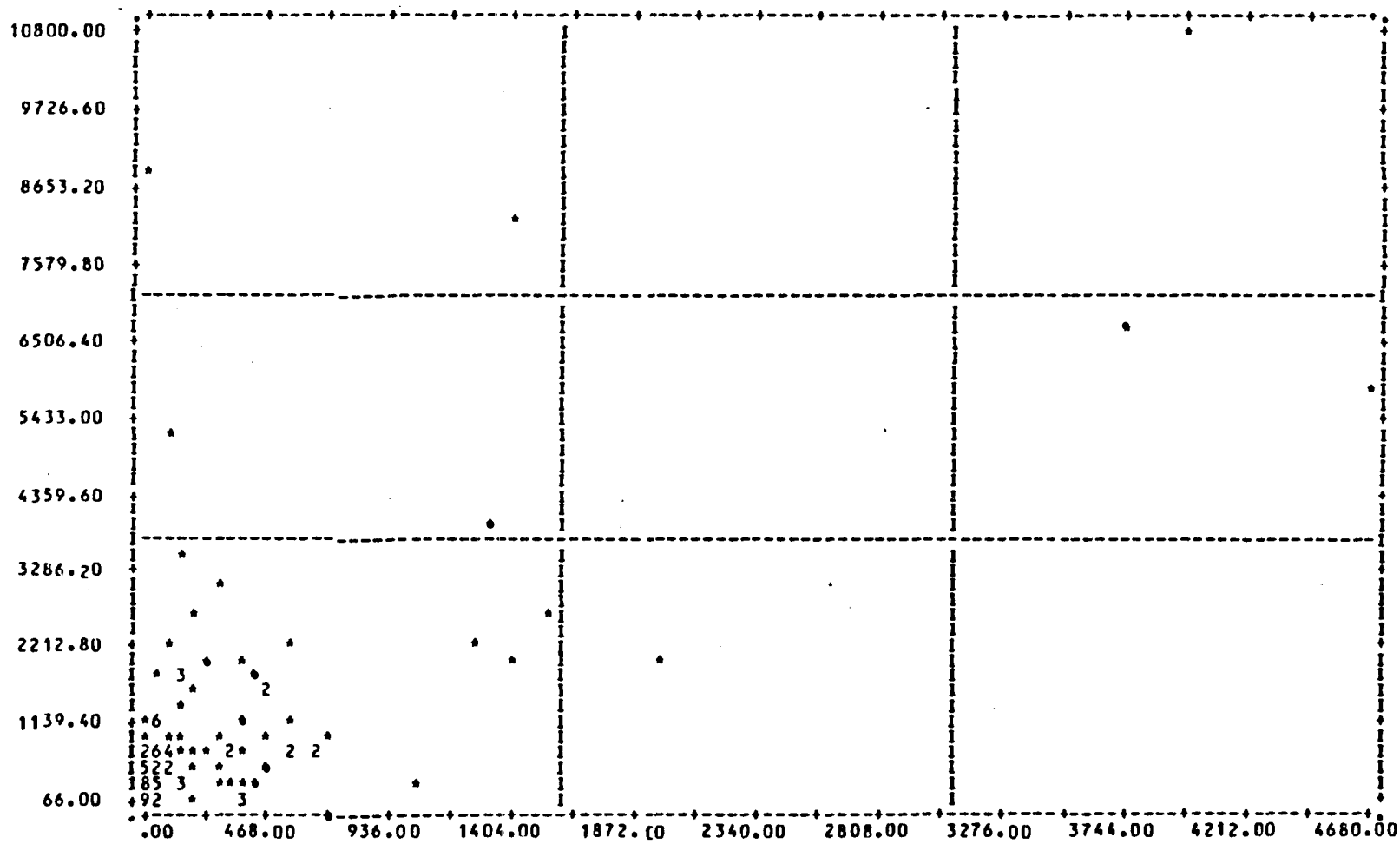


Figure 2

84% accuracy level

Correlation (R): .6719

R square (R^2): .4515

Significance: .00001

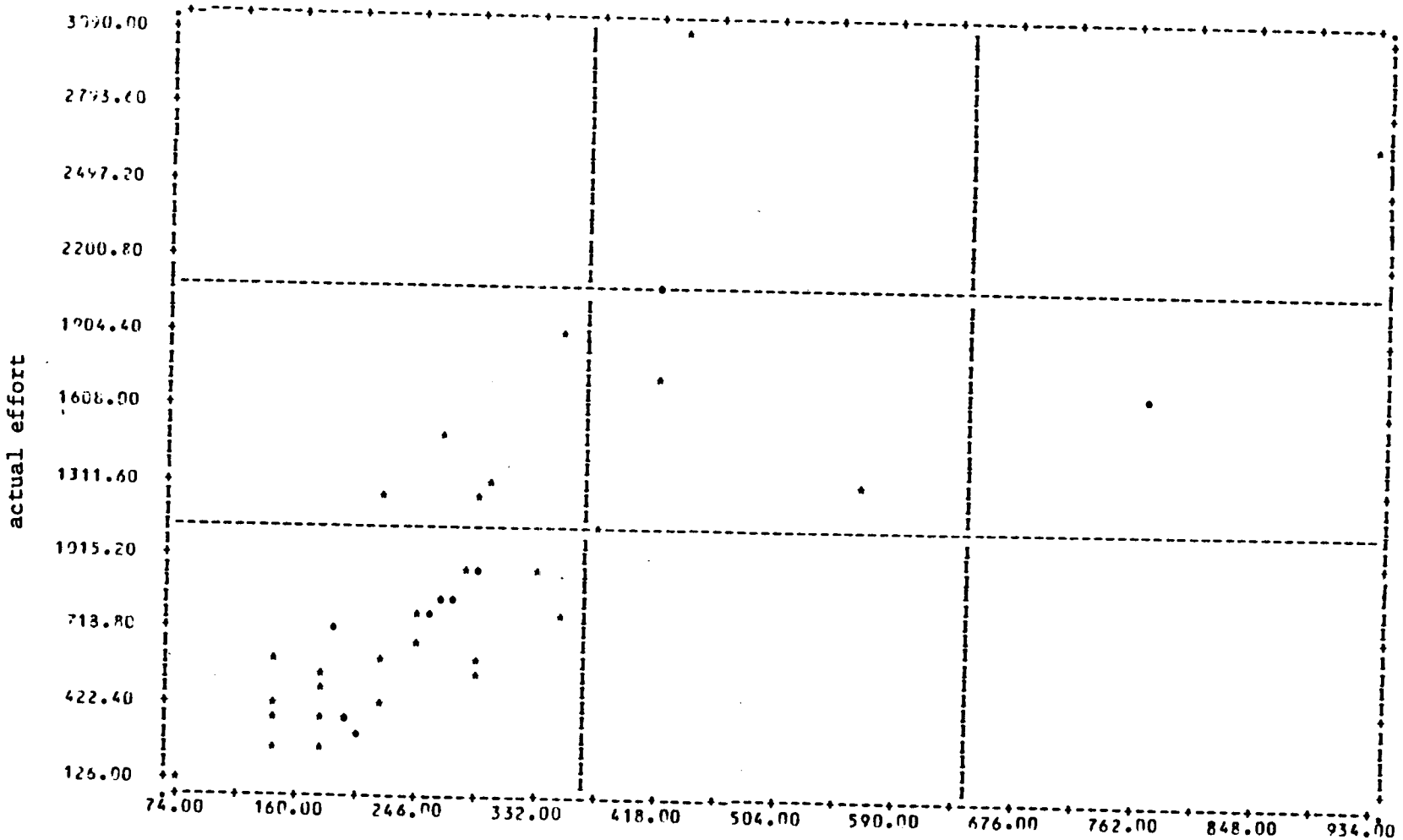


Figure 3

90% accuracy level

Correlation (R): .7583

R square (R^2): .5751

Significance: .00001

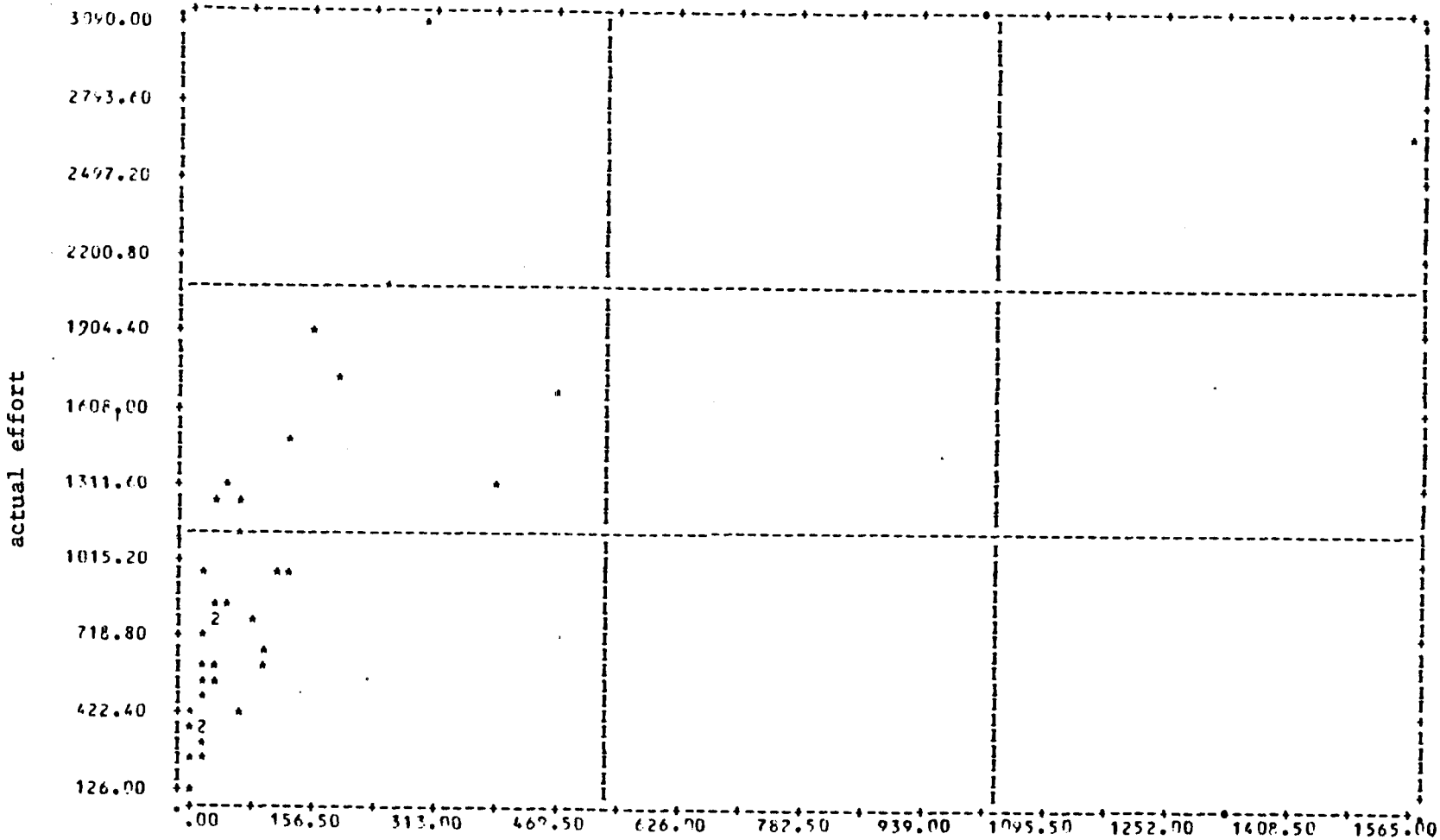


Figure 4

90% accuracy level

Correlation (R): .6612

R square (R^2): .4371

Significance: .00001

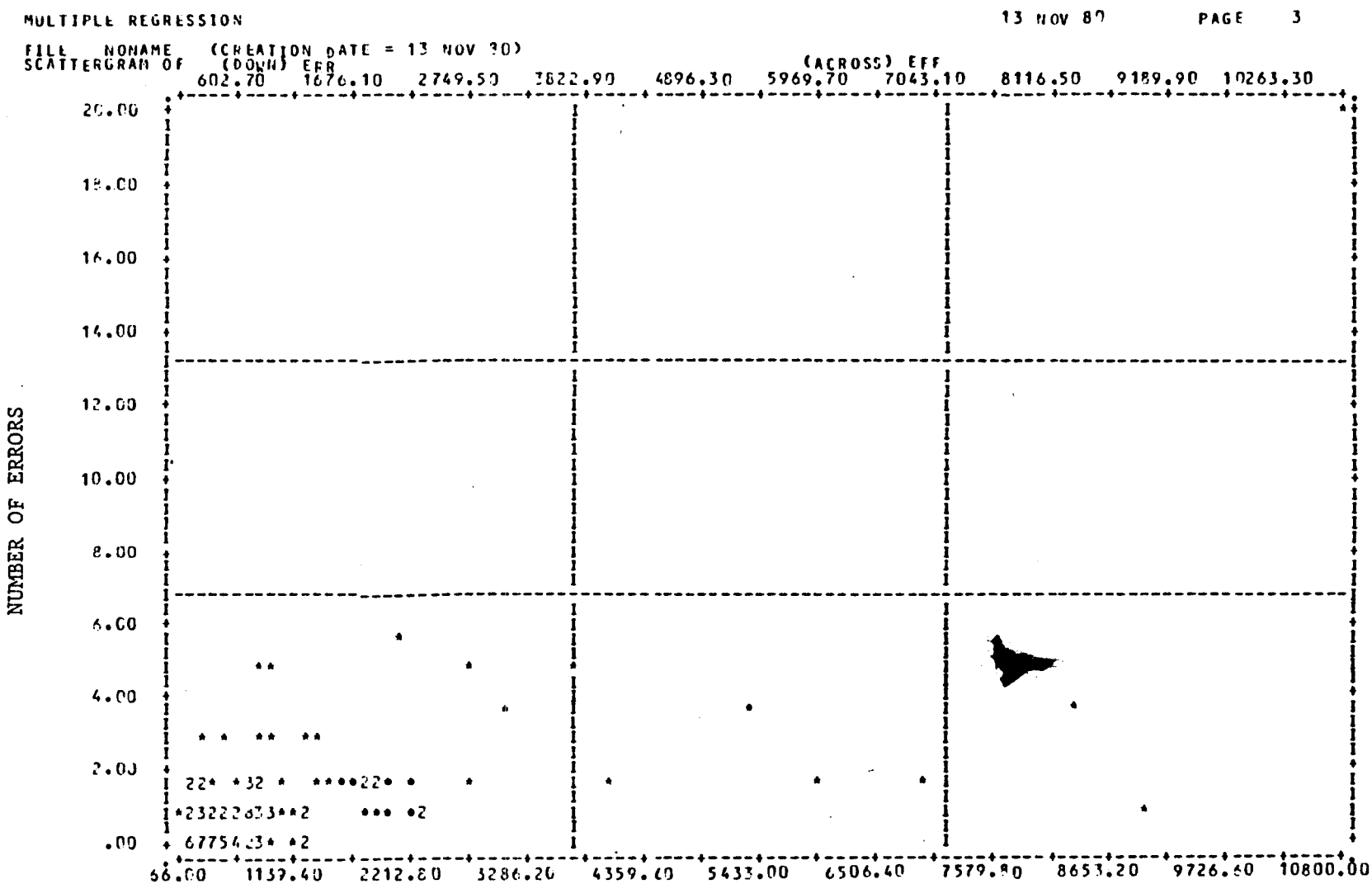


Figure 5

84% accuracy level

Correlation (R): .6227

R square (R^2): .3877

Significance: .00001

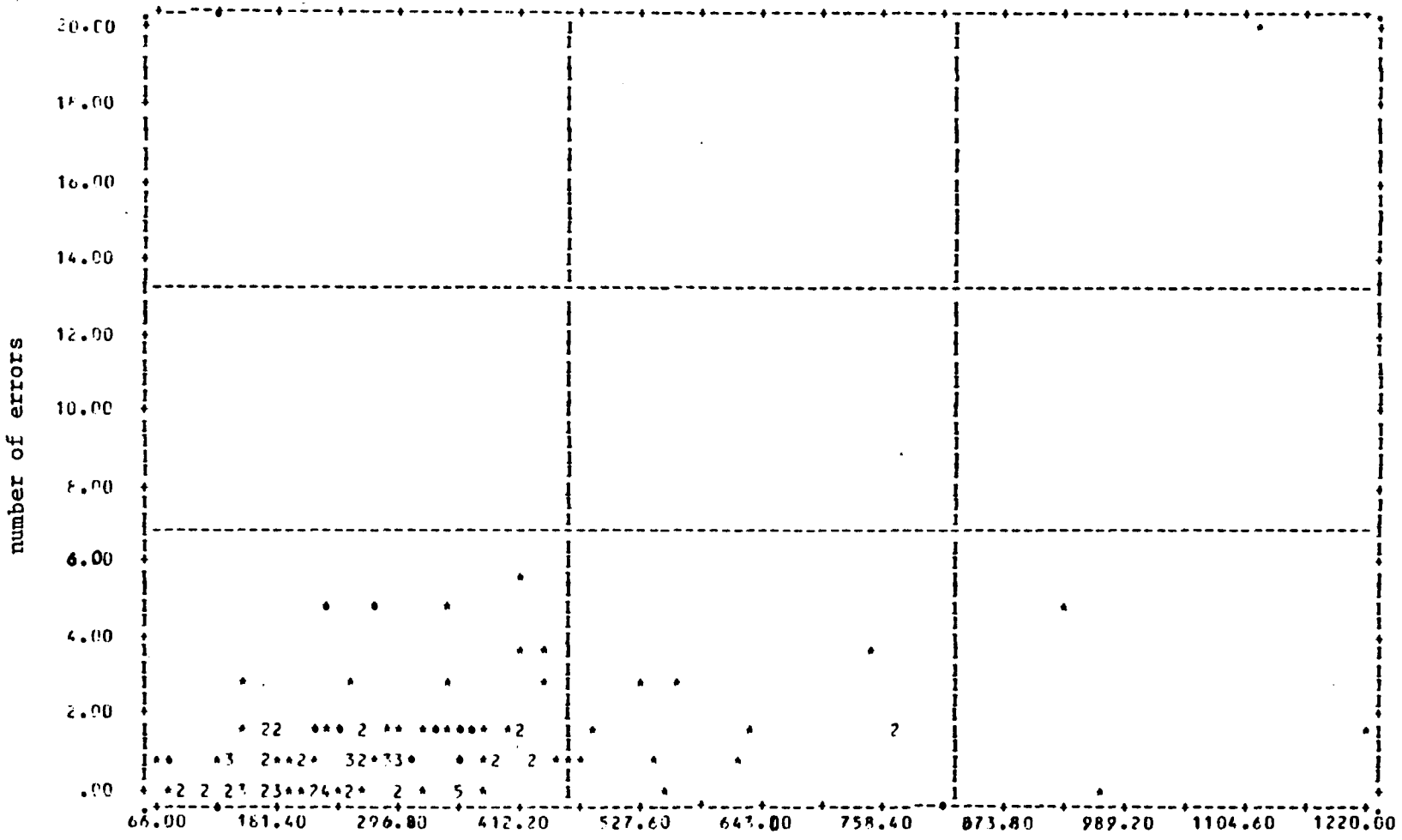


Figure 6

84% accuracy level

Correlation (R): .4890

R square (R^2): .2392

Significance: .00001

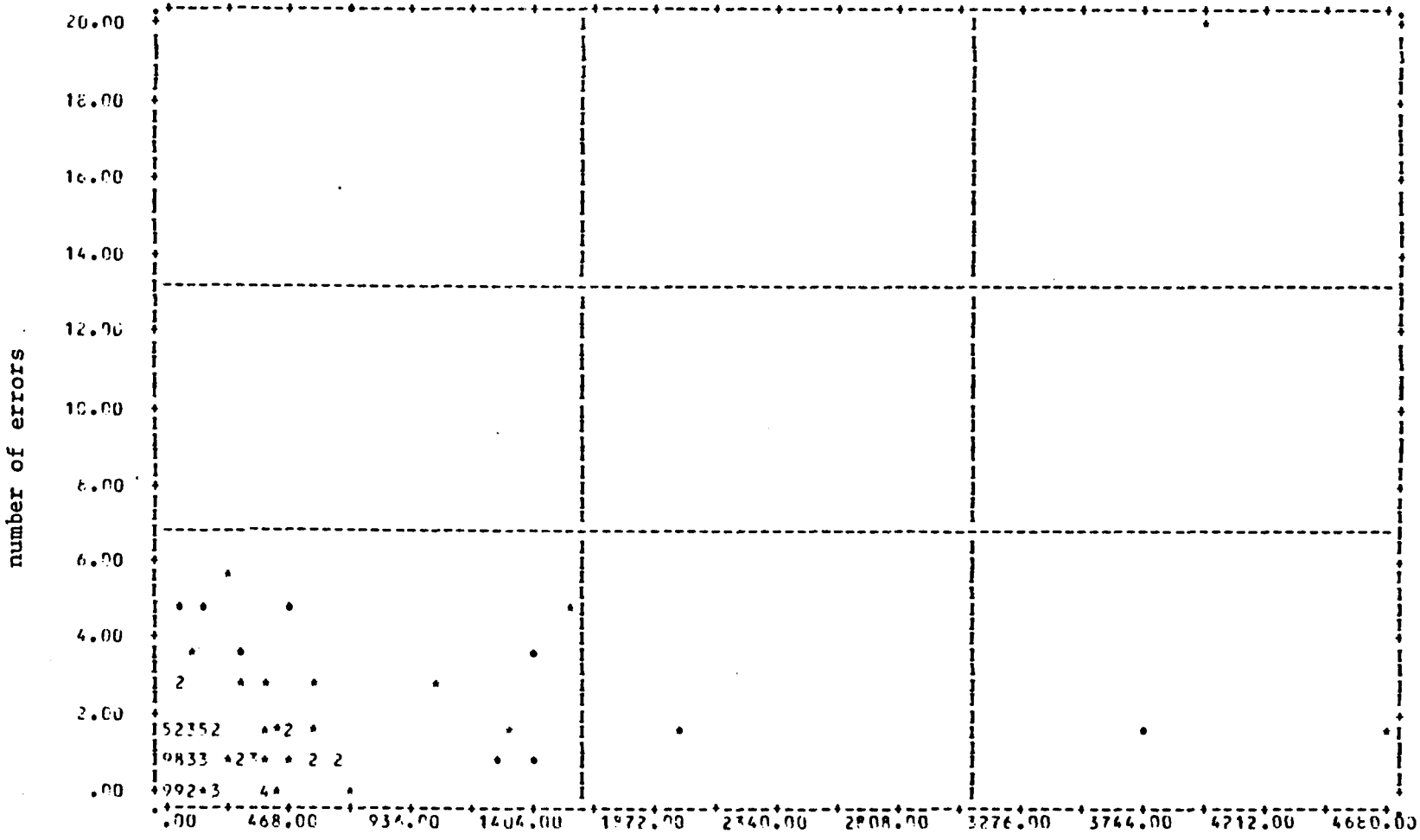


Figure 7

84% accuracy level

Correlation (R): .5028

R square (R^2): .2528

Significance: .00001

SECTION 5 – SOFTWARE ENGINEERING APPLICATIONS

omit

SECTION 5 - SOFTWARE ENGINEERING APPLICATIONS

The technical papers included in this section were originally published as indicated below:

- Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1, copyright 1980 ASME (reprinted by permission of the publisher)
- Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis to Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering, New York: Computer Societies Press, 1981, copyright IEEE (reprinted by permission of the publisher)

R.P D9-61

MODELS AND METRICS FOR SOFTWARE MANAGEMENT AND ENGINEERING

V. R. Basili
University of Maryland
College Park, Maryland

ABSTRACT

This paper attempts to characterize and present a state of the art view of several quantitative models and metrics of the software life cycle. These models and metrics can be used to aid in managing and engineering software projects. They deal with various aspects of the software process and product, including resource allocation and estimation, changes and errors, size, complexity and reliability. Some indication is given of the extent to which the various models have been used and the success they have achieved.

INTRODUCTION

The past few years have seen the emergence of a new quantitative approach to software management and software engineering. It includes the use of models and metrics based on historical data and experience. It covers resource estimation and planning, cost, personnel allocation, computer use, and quality assurance measures for size, structure and reliability of the product.

A quantitative methodology is clearly needed to aid in the software development process. It is needed for understanding and comparison. It was said by Lord Kelvin that if you cannot measure something, then you do not understand it. This is certainly true in the software development domain and is the reason why various models and metrics have been developed, tested, refined and established as aids. One needs models and quantification for comparisons. In cost tradeoffs, for example, it is important to know whether to add another feature, how much an extra level of reliability will cost, or whether a modification to an existing system will be cost effective.

It should be noted, however, that the quantitative approach should augment and not replace good management and engineering judgment. Models and metrics are only tools for the good manager and engineer. This is especially true since the state of the art is newly emerging and not yet well established. Some models and metrics have only been proposed but not fully tested. Others have been tested only in the environment in which they have been developed. However, more and more are being tested and used in environments other than that of the developer. In this paper, some indication of the level of experience with the models or metrics discussed will be given.

Models and metrics must be established via sound testing and experimentation and, before using a model, the manager or engineer should have sufficient knowledge about how much to trust the results of the model. This requires insight into the model, a known confidence level with regard to its reliability and, most important, knowledge of the activity being modeled. None of these models are black boxes and should not be treated as such. Thus, before applying any model, the user should know the nature of his project, whether the assumptions of the model match the environment of his project, and the weaknesses of the model so that he can be careful in evaluating the results.

In what follows, we will cover a large, though by no means exhaustive, set of models. The emphasis will be on those areas where quantitative management can give the greatest payoff. We will discuss process-oriented measures such as size, complexity, and reliability. Each of the measures will be treated to varying degrees. The emphasis will be on categorizing the measures, defining a typical measure or set in the category, and pointing out other measures only when they are different. The references in the back of the paper should help the interested reader pursue a particular measure further or find additional measures not mentioned in this paper.

PROCESS MEASURES

Resources

It is important that we have a better understanding of the software development process and be able to control the distribution of resources such as computer time, personnel, and dollars. We are also interested in the effect of various methodologies on the software development process and how they change the distribution of resources. For this reason, we are interested in knowing the ideal resource allocation, how it may be modified to fit the local environment, the effect of various tradeoffs, and what changes should be made in the methodology or environment to minimize resources expenditure.

There has been a fair amount of work towards developing different kinds of resource models. These models vary in what they provide (e.g., total cost, manning schedule) and what factors they use to calculate their estimates. They also vary with regard to the type of formula, parameters, use of previous data, and staffing considerations. In an attempt to characterize the models, we will define the following set of attribute pairs. Models can be characterized by the type of formula they use to calculate total effort. A single variable model uses one basic variable as a predictor of effort, while a multi-variable model uses several variables. A model may be static with regard to staffing, which means a constant formula is used to determine staffing levels for each activity, or it may be dynamic, implying staffing level is part of the effort formula itself. Within the static multi-variable models, there are various subcategories: adjusted baseline, adjusted table-driven, and multi-parameter equation. The adjusted baseline uses a single variable baseline equation which is adjusted in some way by a set of other variables. An adjusted table-driven model uses a

baseline estimate which is adjusted by a set of variables where the relationships are defined in tables built from historical data. A multi-parameter model contains a base formula which uses several variables. A model may be based upon historical data or derived theoretically. An historical model uses data from previous projects to evaluate the current project and derive the weights and basic formula from analysis of that data. For a theoretical model, the formula is based upon assumptions about such things as how people solve problems. One last categorization is that some models are macro models, which means they are based upon a view of the big picture, while others are micro models in that the effort equation is derived from knowledge of small pieces of information scaled up. We will try to discuss at least one model in each of these categories.

Static single variable models. The most common approach to estimating effort is to make it a function of a single variable, project size (e.g., the number of source instructions or object instructions). The baseline effort equation is of the form

$$EFFORT = a * SIZE^b$$

where a and b are constants. The constants are determined by regression analysis applied to historical data. In an attempt to measure the rate of production of lines of code by project as influenced by a number of product conditions and requirements, Walston and Felix (1) at IBM Federal Systems Division started with this basic model on a data base of 60 projects of 4,000 to 467,000 source lines of code covering an effort of 12 to 11,758 man months. The basic relation they derived was

$$E = 5.2L^{.91}$$

where E is the total effort in man months and L is the size in thousands of lines of delivered source code, including comments. Beside this basic relationship, other relations were defined. These include the relationships between documentation DOC (in pages) and delivered source lines

$$DOC = 49L^{1.01}$$

project duration D (in calendar months) and lines of code

$$D = 4.1L^{.36}$$

project duration and effort

$$D = 2.47E^{.35}$$

and average staff size S (total staff months of effort/duration) and effort

$$S = .54E^{.6}$$

The constants a and b are not general constants. They are derived from the historical data of the organization (in this case, IBM Federal Systems Division). They are not necessarily transportable to another organization with a different environment. For example, the Software Engineering Laboratory (SEL) on a data base consisting of 15 projects of 1.5 to 112 thousand source lines of code covering efforts of 1.8 to 116 staff months have calculated for their environment the following set of equations (2):

$$E = 1.4L^{.94}$$

$$DOC = 29.5L^{.92}$$

$$D = 4.4L^{.267}$$

$$D = 4.4E^{.26}$$

$$S = 2.3E^{.74}$$

Some other variables, including different ways of counting code, were measured by the Software Engineering Laboratory and the equations derived are given here. Letting DL = number of developed, delivered lines of source code (new code + 20% of reused code), M = number of modules, DM = total number of developed modules (all new or more than 20% new) we have

$$E = 1.58DL^{.96}, \quad E = .063M^{1.186}, \quad E = .19DM^{1.0},$$

$$D = 4.6DL^{.28}, \quad D = 2.0M^{.33}, \quad D = 2.5DM^{.3},$$

$$D = 2.0D^{.26}, \quad DOC = 35.7DL^{.92}, \quad DOC = 1.5M^{1.17},$$

$$DOC = 4.8DM^{.99}$$

Most of the SEL equations lie within one standard error of the IBM equation and, since the SEL environment involves the development of more standardized software (software the organization has experience in building), the lower effort for more lines of code seems natural. It is also worth noting that the basic effort vs. lines-of-code equation is almost linear for the SEL--more linear than the Walston/Felix equation. Remember that the project sizes are in the lower range of the IBM data. Lawrence and Jeffery (3) have studied even smaller projects and discovered that their data fits a straight line quite well, i.e., their baseline effort equation is of the form

$$EFFORT = a * SIZE + b$$

where again a and b are constants derived from historical data. The implication here is that the equation becomes more linear as the project sizes decrease.

Static multi-variable models. Another approach to effort estimation is what we will call the static multi-variable model. A resource estimate here is multi-variable because it is based on several parameters, and static because a single effort value is calculated by the model formula. These models fall into several subcategories. Some start with the baseline equation just discussed based on historical data and adjust the initial estimate by a set of variables which attempt to incorporate the effects of important product and process attributes. In other models, the baseline equation itself involves more than one variable.

The models in the adjusted baseline class differ in the set of attributes that they consider important to their application area and development environment, the weights assigned to the attributes, and the constants of the baseline equations.

Walston and Felix (1) calculated a productivity index by choosing 29 variables that showed a significantly high correlation with productivity in their environment. It was suggested that these be used in estimating and were combined in a productivity index

$$I = \sum_i w_i \ x_i$$

where I is the productivity index, w_i is a factor weight based upon the productivity change for factor i and x_i = +1, 0, or -1, depending on whether the factor indicates increased, nominal or decreased productivity.

One model that fits into the single-parameter baseline equation with a set of adjusted multipliers is the model of Boehm (4), whose baseline effort estimate relies only upon project size. His set of attributes are grouped under four areas: (1) product--required fault freedom, data base size, product complexity, adaptation from existing software; (2) computer--execution time constraint, machine storage constraint, virtual machine volatility, computer response time; (3) personnel--analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience; (4) project--modern programming practices, use of software tools, required development schedule. For each attribute Boehm gives a set of ratings ranging from very low to very high and, for most of the attributes, a quantitative measure describing each rating. The ratings are meant to be as objective as possible (hence the quantitative definitions), so that the person who must assign the ratings will have some intuition as to why each attribute could have a significant effect on the total effort. In two of the cases where quantitative measures are not possible, required fault freedom and product complexity, Boehm provides a chart describing the effect on the development activities or the characteristics of the code corresponding to each rating. Associated with the ratings is a chart of multipliers ranging from about .1 to 1.8. Another model which falls into this category is the model of Doty (5). The Doty model, however, provides a different set of weights for different applications besides two ways to estimate size.

One model which falls into the category of adjusted table-driven is that of Wolverton (6). Here the basic algorithm involves categorizing the software routines. The categories include control, I/O, pre- or post-algorithm processor, algorithm, data management, and time critical routines. Each of these routines has its own cost-of-development curve, depending upon the degree of difficulty (easy, medium, or hard) and the newness of the application (new or old). The cost is then the number of instructions by category and degree of difficulty times the corresponding cost taken from a table. Another model of this type, but more simplistic, is Aron (7).

The GRC model (8) involves a set of equations derived from historical data and theory for the various activities, several of which are multi-parameter equations of more than one variable. For example, the equation for code development is

$$MM_{CD} = .9773 \times N_{OF}^{1.2583} \times e^{-.08953 * Y_{EXP}}$$

where MM_{CD} is the baseline staff months for code development task group for a subsystem, N_{OF} = the number of output formats for a subsystem and Y_{exp} is the average years of staff experience in code development. It is worth noting that size of the code is not a factor in this formula. Other formulas exist for the effort involved in analysis and design, system level testing, documentation, installation, training, project control, elapsed time and a reasonable check for the total staff months for the project (MM_{PROJ})

$$MM_{PROJ} = .0218 * ((2 + N_{OF}) * \ln(2 + N_{OF}))^{1.71}$$

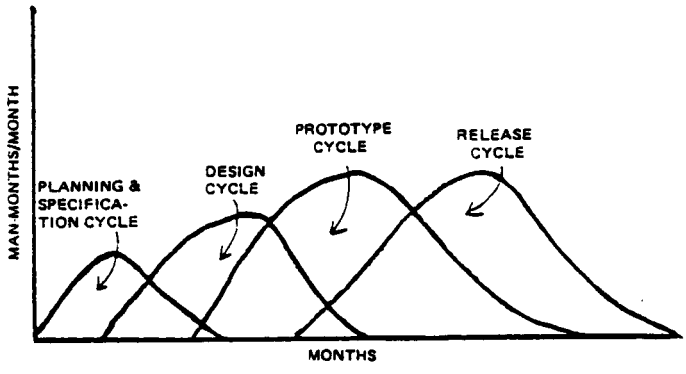
where N_{OF} is as defined above.

Dynamic multi-variable models. Once an effort estimate is made, the next question of concern is how to assign people to the project so that the deadlines for the various development activities will be met. Here again there are basically two approaches: the one empirical, the other theoretical. Each of the methods discussed so far uses the empirical approach which tries to identify the activities which are a part of the development process of a typical project for their software house. Then, using accounting data from past projects, they determine what percentage of the effort was expended on each activity. These percentages serve as a baseline and are intuitively adjusted to meet the expected demands of a new project. For example, in the Wolverton model, total cost is allocated into five major subareas: analysis cost (20% of total), design cost (18.7% of total), coding cost (21.7% of total), testing cost (28.3% of total) and documentation cost (11.3% of total). Each of these subarea costs are subdivided again, depending upon the activities in the subareas. In this way, each activity can be staffed according to its individual budget. Allocation of time is determined by history and good management intuition.

The theoretical approach attempts to justify its resource expenditure curve by deriving it from equations which model problem-solving behavior. In other words, the resource model lays out the staffing across time and within phases. We will refer to this approach as the dynamic multi-variable model. It is dynamic because the model produces a curve which describes the variation of staffing level across time. The model is multi-variable because it involves more than one parameter.

Two models in this category will be discussed which differ in the assumptions they make. The first model, which is the most widely known and used, is the Putnam model (9).

The model is based on a hardware development model (10) which noted that there are regular patterns of manpower buildup and phase-out independent of the type of work done. It is related to the way people solve problems. Thus, each activity could be plotted as a curve which grows and then shrinks with regard to staff effort across time. For example, the cycles in the life of a development engineering project look as follows:



Similar curves were derived by Putnam for software cycles which are: planning, design and implementation, testing and validation, extension, modification and maintenance.

The theoretical basis of the model is that software development is a problem-solving effort and design decision-making is the exhaustion process. The various development activities partition the problem space into subspaces corresponding to the various stages (cycles) in the life cycle. A set of assumptions is then made about the problem subset: (1) the number of problems to be solved is finite, (2) the problem-solving effort makes an impact on and defines an environment for the unsolved problem set, (3) a decision removes one unsolved problem from the set (assumes events are random and independent) and (4) the staff size is proportional to the number of problems "ripe" for solution. Because the model is theoretically based (rather than empirically based) some motivation for the equation is given. Consider a set of independent devices under test (unsolved problem set) subject to some environment (the problem-solving effort) which generates shocks (planning and design decisions). The shocks are destructive to the devices under test with some dependent conditional probability distribution p(t) which is random and independent with some rate parameter λ. Assume the distribution is Poisson and let T be a random variable associated with the time interval between shocks

$$Pr(T > t) = Pr \quad (1)$$
(no event occurs in interval (o, t))

where t = o is the time of the most recent shock letting p(t) be the conditional probability of a failure given that a shock has occurred and λ be the Poisson rate parameter, then

$$Pr(T > t) = e^{-\lambda \int_0^t p(x)\,dx} \quad (2)$$

and

$$Pr(T \leq t) = 1 - e^{-\lambda \int_0^t p(x)\,dx} \quad (3)$$

and the p.d.f. associated with (3) is

$$f(t) = \lambda * p(t) * e^{-\lambda \int_0^t p(x)\,dx},\ t \geq o$$

This leads to the class of Weibull distributions (known in reliability work) with the physical interpretation that the probability of devices succumbing to destructive shocks is changing with time. Based upon observed data on engineering design projects, a special case of (3) can be used

$$y = f(t) = 1 - e^{-at^2} \quad (4)$$

where $p(t) = \alpha t$ (5)

and $a = \frac{\lambda \alpha}{2}$ (6)

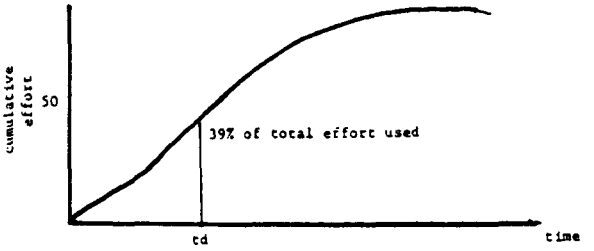
Note that this implies engineers learn to solve problems with an increasing effectiveness (i.e., familiarity with the problems at hand leads to greater insight and sureness). Parameter a consists of an insight generation rate λ and a solution finding factor α. Equation (5) is a special linear case of the family of learning curves: $y = a\,x^b$.

Equation (4) is then the normalized form of the life cycle equation. By introducing a parameter (K) xpressed in terms of effort, we get an effort curve, the integral form of the life cycle equation

$$y = K * (1 - e^{-at^2})$$

where

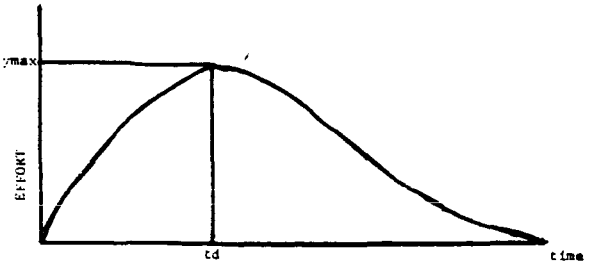
- y is the cumulative manpower used through time t
- K is the total manpower required by the cycle stated in quantities related to the time period used as a base, e.g., man-months/month
- a is a parameter determined by the time period in which y´ reaches its maximum value (shape parameter)
- t is time in equal units counted from the start of the cycle



The life cycle equation (derivative form) is

$$y' = 2\,K\,a\,t\,e^{-a\,t^2}$$

where y´ is the manpower required in time period t stated in quantities related to the time period used as a base and K is the total manpower required by the cycle stated in the same units as y´.



The curve (called the Rayleigh Curve) represents the manpower buildup. The sum of the individual cycle curves results in a pure Rayleigh shape. Software development is implemented as a functionally homogenous effort (single purpose). The shape parameter a depends upon the point in time at which y´ reaches its maximum, i.e.

$$a = {}^{1}/_{2t_d^2}$$

where t_d is the time to reach peak effort. Putnam has empirically shown t_d corresponds closely to the design time (time to reach initial operational capability). Substituting for a we can rewrite the life cycle equation as

$$y' = \frac{K}{t_d^2} * t e^{-t^2/2\,t_d^2}$$

The equations given are for the entire life cycle. To find development effort only

take

$$y = K * (1-e^{-at^2})$$

substitute $a = 1/2t_d^2$

$$y = K * (1-e^{(-t^2/2t_d^2)})$$

then the development effort is time to t_d

$$y = K * (1-e^{(-t^2/2t_d^2)})$$

$$= K * (1-e^{-.5})$$

$$= .3935K$$

or DE = 40% of LC effort

The life cycle and development costs may be calculated by multiplying the cost for that cycle by staff year cost

$$\$LC = K*MC$$

where MC = mean cost (in $) per man year of effort

K = total manpower (in man years) used by the project

(Note: the equation neglects computer time, inflation, overtime, etc.)

and

$$\$DEV = MC * (.3935K) \approx .4 * \$LC$$

Putnam found that the ratio $K/(t_d^2)$ has an interesting property. It represents the difficulty of a system in terms of programming effort required to produce it. He defines

$$D = K/(t_d^2)$$

To illustrate how management decisions can influence the difficulty of a project, assume a system size of K = 400 MY and t_d = 3 years. Then the difficulty D = 400 / 9 = 44.4 man years per year squared.

Consider a management decision to cut the life cycle cost of the system by 10%. Now, K = .9 * (400) = 360 MY and D = 360 / 9 = 40. This results in a 10% decrease in assumed difficulty of the project. This decision assumes the difficulty is less than it really is, and the result is less product.

Now consider the more common case of attempted time compression. Assume management makes a decision to limit the expended effort to 400 MY, but wants the system in 2.5 years instead of 3 years. Now, K = 400 MY, t_d = 2.5 years, and D = 400 / 6.25 = 64 (a 44% increase). The result of shortening the natural development time is a dramatic increase in the system difficulty.

The Putnam model generates some interesting notions. Productivity is related to the difficulty and the state of technology; management cannot arbitrarily increase productivity nor can it reduce development time without increasing difficulty. The tradeoff law shows the cost of trading time for people.

In deriving an alternate model, Parr (11) questions the assumption of the Rayleigh equation that the initially rising work rate is due to the linear learning curve which governs the skill available for solving problems. He argues that the skill available on a project depends on the resources applied to it and that the assumption confuses the intrinsic constraints on the rate at which software can be developed with management's economically-governed choices about how to respond to these constraints.

As an alternative to this assumption, his model suggests that the initial rate of solving problems is governed by how the problems in the project are related, i.e., the dependencies between them. For example, the central phase of development is naturally suited to rapid rates of progress since that is when the largest number of problems are visible. Letting V(t) be the expected size of this set of visible (available for solving) problems at time t, Parr's model yields the equation

$$V(t) = \frac{\alpha A e^{-\gamma \alpha t}}{(1 + A e^{-\gamma \alpha t})^{(\gamma + 1/\gamma)}}$$

where

- α is the proportionality constant relating the rate of progress and the expected size of the visible set
- A is a measure of the amount of work done on the project before the project officially starts
- γ is a structuring index which measures how much the development process is formalized and uses modern techniques.

The curve represented by V(t) differs from the Rayleigh/Norden curve for y'(t) in two important ways. The Rayleigh curve is constrained to go through the origin; the Parr curve is not. Making y'(0) = 0 corresponds to setting an official start date for the project. Before that point, the effort expended on the project is assumed to be minimal. In reality, there is often a good deal of work done before that date, including such activities as requirements analysis and feasibility studies. In Putnam's environment, these were handled by a separate organization and could be ignored. Another factor that affects the problem space is past experience in the application area, or even more tangible is the influence of design or code taken from past projects. All of these have the effect of structuring the problem space at the beginning, so that more progress can be made early. The Parr curve accounts for this; the Putnam curve does not. See Fig.1 for a comparison of the two curves.

A second distinction between the two curves is the flexibility of where the point of maximum effort can come. By using a structuring index greater than one, this point of maximum effort can be delayed almost to acceptance testing and effort could still be drastically reduced before project completion. With the Rayleigh curve, a late point of maximum effort constrains the curve to have a slow buildup and almost no decay at the end.

Parr does not say how to estimate the parameters for V(t) in terms of data the project manager would have on hand. This is a problem in doing resource estimation currently, but the model could use the existing resource allocation schedule, based on early data points, to predict the latter part of the curve. The Parr model is only currently being tested on real

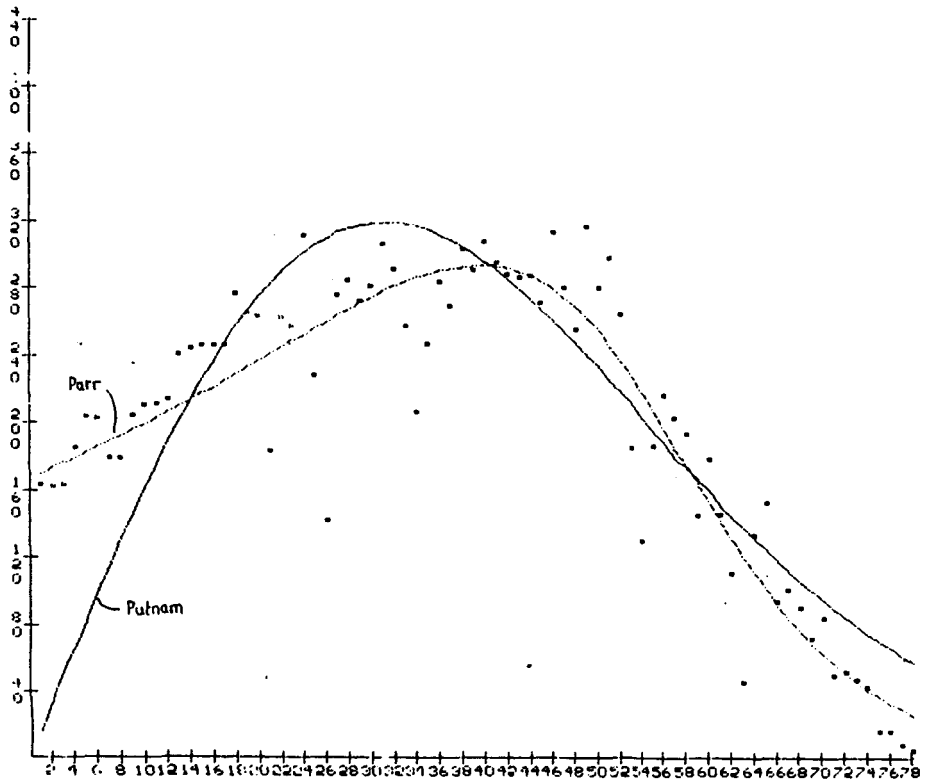


Fig.1 Weekly direct labor resources (manhours)
PARR vs PUTNAM

software for the first time and the results are not yet available. The Rayleigh model, on the other hand, has been used in many environments and has been quite successful on the whole.

Single variable, theoretical. The two previous theoretical models may be thought of as macro models in that the estimate of staffing levels relies on process oriented issues, such as total effort, schedule constraints, and the degree that structured methodology is used. Product oriented issues, such as source code, are not a factor. Most of the other models are less macro oriented in that they consider product characteristics, such as lines of code and input/output formats. In this section, we will discuss another type of theoretical model, based upon lower level aspects of the product, which we will call a micro model. The particular model discussed here deals with the idea that some basic relationships hold with regard to the number of unique operators and operands used in solving a problem and the eventual effort and time required for development. This notion was proposed by Halstead as part of his software science (12). Here there is only one basic parameter--size--measured in terms of operators and operands. The model transcends methodology and environmental factors. Most of the work in this area has dealt with programs or algorithms of module size rather than with entire systems, but that appears to be changing.

In the language of software science, measurable properties of algorithms are

n_1 number of unique or distinct operators in an implementation

n_2 number of unique or distinct operands in an implementation

$f_{1,j}$ number of occurrences of the j^{th} most frequent operator, $j = 1, 2, ..n_1$

$f_{2,j}$ number of occurrences of the j^{th} most frequent operand, $j = 1, 2, ..n_2$

then the vocabulary of an algorithm is

$$n = n_1 + n_2$$

and the implementation length is

$$N = N_1 + N_2$$

where

$$N_1 = \sum_{j=1}^{n_1} f_{1,j} \quad , \quad N_2 = \sum_{j=1}^{n_2} f_{2,j} \quad , \quad N = \sum_{i=1}^{2} \sum_{j=1}^{n_i} f_{i,j}$$

Based only on the unique operators and operands, the concept of program length N can be estimated as

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

$\hat{N}$ is actually the number of bits necessary to represent all things that exist in the program at least once, i.e., the number of bits necessary to represent a symbol table. Over a large set of programs in different environments, it has been shown that $\hat{N}$ approximates N very well.

To measure the size of an algorithm, software science transcends the variation in language and character set by defining algorithm size (volume) as the

minimal number of bits necessary to represent the implementation of the algorithm. For any particular case, there is an absolute minimum length for representing the longest operator or operand name expressed in bits. It depends upon n, e.g., a vocabulary of 8 elements requires 8 different designators, or $\log_2 8$ is the minimal length in bits necessary to represent all individual elements in a program. Thus, a suitable metric for size of any implementation of any algorithm is $V = N \log_2 n$, called volume.

The most succinct form in which an algorithm can be expressed requires a language in which the required operation is already defined and implemented. The potential volume, V*, is defined as

$$V^* = (N_1^* + N_2^*) \log_2 (n_1^* + n_2^*)$$

but minimal form implies $N_1^* = n_1^*$ and $N_2^* = n_2^*$ because there should be no repetition. The number of operators should consist of one distinct operator for the function name and another to serve as an assignment or grouping symbol so $n_1^* = 2$. Thus, $V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$ where n_2^* represents the number of different input/output parameters. Note: V* is considered a useful measure of an algorithm's content. It is roughly related to the basic GRC model concept of input/output formats. In fact, the GRC equation for man months of the project (MM_{PROG}) is an exponential relationship between MM_{PROG} and an estimate of V*.

The level of the implementation of a program is defined as its relation to its most abstract form, V*, i.e., $L = \frac{V^*}{V}$. $L \leq 1$ and the most succinct expression for an algorithm has a level of 1. $V^* = L \times V$ implies that when the volume goes up the level goes down. Since it is hard to calculate V*, an approximation for L, $\hat{L}$, is calculated directly from an implementation

$\hat{L} = \frac{2n_2}{n_1 N_2} \sim L$. The reciprocal of level is defined as the difficulty, $D = 1/L$, which can be viewed as the amount of redundancy within an implementation.

Based on these primitives, formulas for programming effort (E) and time (T) are derived. Assuming the implementation of an algorithm consists of N selections from a vocabulary of n elements and that the selection is non-random and of the order of a binary search (implying $\log_2 n$ comparisons for the selection of each element), the effort required to generate a program is $N \log_2 n$ mental comparisons (this is equal to the volume (V) of the program). Each mental comparison requires a number of elementary mental discriminations where this number is a measure of the difficulty (D) of the task. Thus, the total number of elementary mental discriminations E required to generate a given program should be $E = V * D = V/L = V^2/V^*$. This says the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language. E has often been used to measure the effort required to comprehend an implementation rather than produce it, i.e., E may be a measure of program clarity.

To calculate the time of development, software science uses the concept of a moment, defined by the psychologist Stroud as the time required by the human brain to perform the most elementary discrimination. These moments have been shown to occur at a rate of 5 to 20 per second. Denoting moments (or Stroud's number) by S, we have $5 \leq S \leq 20$ per second. Assuming a programmer does not "time share" while solving a problem, and converting the effort equation (which has dimensions of both binary digits and discriminations) we get $T = \frac{E}{S} = \frac{V}{SL} = \frac{V^2}{SV^*}$. Halstead empirically estimated $S = 18$ for his environment, but this may vary from environment to environment.

Software science metrics have been validated in a variety of environments but predominantly for module size developments.

Other resources. In what has been stated so far, resource expenditure and estimation have been predominantly computed in terms of effort. The formula for cost may be a simple multiplication of the staff months times the average cost of a staff member or it may be more complicated. It may include some difference for the cost of managers versus the cost of programmers versus the cost of support personnel whose role varies across the life cycle (13).

The schedule may be derived based upon historical data, with effort allocated to different activities based upon the known percentages or it may be dictated by the model itself, as with the Rayleigh curve. However, the dynamic models generate what they consider the ideal staffing conditions which may not be the actual ones available. Thus, in fitting actual effort to the estimated or proposed effort, some decisions and tradeoffs must be made.

Computer time is yet another resource. Unfortunately, none of the above models treats this within the same formula. In general, they have a separate formula for computer time again based upon computer use in similar projects. These models vary from a simple table type model (6) to some very sophisticated probability distribution based on reliability modeling for phases of the development, such as testing (14).

Changes and Errors

There are process aspects other than resource expenditures that provide information about managing and engineering the process and the product. One such aspect is the changes and errors generated during development or maintenance. Monitoring the changes in the software provides a measure of level of effort to get the product in order. If we can classify the types of changes that occur or their source of origin, we can categorize the environment and gain insight into how to manage or minimize the effect of particular types of changes. For example, suppose the user is generating a series of major changes at a continual rate. This may provide management with the information it needs to reclassify the environment from its original one to a more complex one, permitting modification of the cost parameters in the resource estimation model and a re-estimation of cost part way through the project. It could also provide management with the necessary insight to change the development approach or methodology to one that is more insensitive to externally generated change, such as some incremental development approach.

Monitoring errors provides information with regard to the quality of the product. A product developed with only a few errors or with errors found early and an error rate decreasing during development and testing will warrant more confidence in its quality. Keeping track of the time to find and fix errors gives insights into cost. Knowing the types of errors being made helps in focusing attention to particular problems during the code-reading and design-review sessions.

Program evolution measures. Belady and Lehman (15) have examined the changes occurring in software during maintenance and derived a set of laws for program evolution. Based on such parameters as size of the system, number of modules added, deleted or changed, the release data, manpower, machine time and cost, they derived the following laws:

1. Law of continuing change. A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

2. Law of increasing entropy. The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

3. Law of statistically smooth growth. Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self-regulating with well-defined long-range trends.

These laws can be demonstrated by using the following metrics:

RSN, the release number

D_r, the age of system at release R

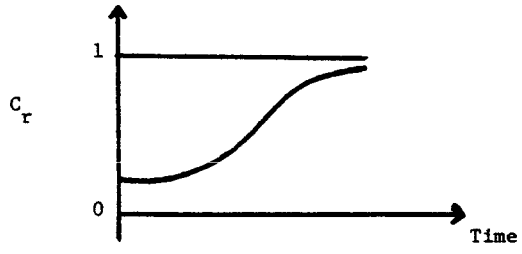
I_r, the time between releases R-1 and R

M_r, the number of modules in the system

MH_r, the number of modules handled during release interval I_r (estimator of activity undertaken in each release)

$HR_r = MH_r/I_r$, the handle rate

$C_r = MH_r/M_r$, the complexity which is the fraction of released system modules that were handled during the course of the release R.



C_r has been observed to be monotonically increasing and approaching unity over time (for OS 360, approximately 20 releases over 10 years).

Using these metrics, management can predict when it is too costly to modify a system, i.e., when it is cheaper to redesign than make the next change. It can also determine whether enough effort is being devoted to keep future changes at a reasonable cost.

Program-changes. Dunsmore and Gannon have proposed a measure called program-changes which correlates very highly with errors (16). A program-change is a textual revision in the source code of a module during the development period. One program-change should represent one conceptual change to the program. Thus, a program-change is defined as one or more changes to a single statement, one or more statements inserted between existing statements, or a change to a single statement followed by the insertion of new statements. On the other hand, the following are not counted as program-changes: the deletion of one or more existing statements, insertion of standard output statements or special compiler-provided debugging directives, and insertion of blank lines or comments. Basili and Reiter showed that program-changes were minimal when a good software development method was used (17).

Error-day. An error-based measure of product quality was proposed by Mills (18) which he called the error-day. The motivation is that the longer an error remains in the system the more expensive and less reliable it is. The error-day measure is simply the sum over each error of the number of days it has existed within a system. It weights errors by their duration in the system. Clearly, a low error-day count is an indicator of a well-engineered program. This measure could be automated by using the concept of program-changes and plotting them against time.

Job-steps. An indication of the amount of effort expended in development can be the number of computer accesses or job-steps. A computer job-step is a single programmer-oriented activity performed on a computer at the operating system command level, which is basic to the development effort and involves nontrivial expenditures of computer or human resources. Typical job-steps might be text editing, module compilation, link editing, and program execution. Basili and Reiter (17) found job-steps to be a serious differentiator of development environments, and that good methodology leads to a smaller number of job-steps.

There exist many other measures of the software development process. The interested reader is referred to some general references in the literature, e.g., Curtis (19), Mohanty (20), Belady (21).

PRODUCT MEASURES

Actually, all the previous measures could have been considered measures of the product. If a product takes a long time or a large effort to develop, we may consider it a complex product. If there were lots of errors found at the tail end of product development or if the rate of finding errors was increasing every day, we would say the quality of the product was very low. However, each of those indicated as much, if not more, about the process than the product.

The measures discussed in this section are probes into the product. They are taken at a discrete point in time, usually on the final deliverable product. Even though examining the changes in value of the metrics on the product over time could be very informative with regard to the process, we will classify them as product measures. We categorize these measures with respect to size, structure and reliability.

Size

The size of a product is a simplistic measure and easy to calculate. It is a reasonable indicator of the amount of work expended and correlates well with effort. Size metrics are used for cost estimation, comparison of products, and for measures of productivity. Although it may be a basic ingredient in effort and productivity measures, it must be modified by many other factors, such as reliability and complexity. These measures will be treated in subsequent sections.

The most common measure of size is lines of code.

However, what gets measured depends to a great extent on our interests. For example, if we are interested in measuring effort, then source lines including comments and data are a reasonable measure and have been used in several studies (1, 2). If we are interested in function size, a better approximation may be executable statements. If our interest is in comparing the size of resulting products for operational use, a common denominator is number of machine language instructions. Clearly, there is little agreement on the appropriate measure of lines of code and the choice should depend upon the issue under consideration. It is important in reading the literature that we clearly understand which measure of size is being used, since the authors do not always make it clear.

Another measure of size is to treat units larger than lines of code. One common unit is the module. Modules are used in the measures of Belady and Lehman (21) and were shown to be reasonable measures for cost estimation by Freburger and Basili (2). Smaller units, such as procedures or functions, were used by Basili and Reiter (17). Again, the choice is dependent upon the purpose of the measure. For estimation, it is sometimes easier to predict the number of modules rather than the number of lines. However, comparison may be difficult since there is no standard definition of module.

On the other end of the size spectrum is the number of operators and operands as defined in software science by Halstead (12). More specifically, the length and volume measures are potential measures for size of an implementation and size of the function, respectively. There have been several studies that support these metrics as reasonable approximations to what they purport to measure. They make good metrics for comparison and possible evaluation, but there is potential for using them for estimation also.

Structure

The structure of a program is often a good indicator of whether that product is well designed, understandable, and easy to modify. Structure measures are often proposed as measures of the complexity of the product. In examining structure, we may be concerned with the control structure, the data structure, or a mixture of the two.

Control structure measures. The simplest control structure metric is the number of decisions (17) as measured by the number of constructs that represent branches in the flow of control, such as if then else or while do statements. There is a basic belief that the more control flow branching there is in a system the more complex it is. A variation of this measure is the relative percentage of control flow branching, i.e., the number of decisions divided by the number of executable statements. Early studies by Aron (7) showed that varying levels of this type of complexity could account for a nine to one difference in productivity.

A more refined measure of control complexity is cyclomatic complexity as proposed by McCabe (22). The cyclomatic complexity of a graph is defined as the number of edges minus the number of nodes plus the number of connected components, and is equal to the minimum number of basic paths from which all other paths may be constructed. Given a program in which all statements are on a path from the entry node to an exit node, the cyclomatic complexity can be defined as the number of predicates plus the number of segments. A predicate is defined as a simple Boolean expression governing the flow of control and a segment is defined as an individual routine (procedure or function).

The measure originated as a count of the minimum number of program paths to be tested. This is one quantitative measure of a program's complexity. The measure is usually applied at the module level and McCabe proposed a cyclomatic complexity of ten as an upper bound for the safe range with regard to the complexity of a module. Several variations of the basic cyclomatic complexity measure have been studied by Basili and Reiter (23). They evaluated their sensitivity to different software development environments with reasonable success. They have also defined some approaches to using the measure at the product level rather than the module level in a way that is reasonably insensitive to system modularization.

Other measures of control complexity involve the weighting of various types of control structures as to whether they are simple or complex, where simple means easy to read and prove correct based upon the graph structure. For example, single-entry single-exit program graphs that contain a single predicate node are easier to understand and abstract from than more complicated graph structures. Thus, one approach would be to weight various graph structures based upon this complexity. This type of measure requires a more detailed analysis of the program structure than does the cyclomatic complexity measure, but tends to be a deeper measure of control flow and can include other complexity factors, such as nesting level. One such measure is essential complexity (22), which assigns every program using only structured programming control structures a complexity of one.

Data structure measures. Data structure metrics try to measure the complexity of the program structure by the way the data is used, organized, and allocated. Clearly, the simpler the reader's ability to abstract the use of data the easier the program will be to understand and modify. Several measures have been used for evaluating the structuring of the data in a program and a few will be discussed here.

The segment-global usage pair metric (24) attempts to measure the goodness of the use of globals in the program. A segment-global usage pair (p, r) is an instance of a global variable r being used by a segment p (i.e., r is either modified or accessed by p. Each usage pair represents a unique "use connection" between a global and a segment. Let actual usage pair (AUP) represent the count of realized usage pairs, i.e., r is actually used by p. Let possible usage pair (PUP) represent the count of potential usage pairs, i.e., given the program's globals and their scopes, the scope of r contains p so that p could potentially modify or access r. This represents a worst case. Then the relative percentage usage pairs (RUP) is RUP = AUP/PUP and is a way of normalizing the number of usage pairs relative to the problem structure. The RUP metric is an empirical estimate of the likelihood that an arbitrary segment uses an arbitrary global.

The data binding metric (24, 25) is an attempt at measuring the inter-relationship of modules or segments within a program. A segment-global-segment data binding (p, r, q) is an occurrence of the following: (1) segment p modifies global variable r, (2) variable r is accessed by segment q, and (3) $p \neq q$. The existence of a data binding (p, r, q) implies that q is dependent on the performance of p because of r. Binding (p, r, q) does not equal binding (q, r, p). (p, r, q)

represents a unique communication path between p and q and the total number of data bindings represents the degree of a certain kind of "connectivity," i.e., between segment pairs via globals, within a complete program. Let actual data bindings (ADB) represent the absolute number of realized data bindings in the program, i.e., the realized connectivity, and possible data bindings (PDB) represent the absolute number of potential data bindings given the program's global variables and their declared scope (i.e., same worst case). Then we can normalize the number of data bindings by calculating the relative percentage RDB = ADB/PDB. This gives some relative measure of the amount of information exchanged in the program.

A measure of the amount of data required to be understood by the programmer while reading a program is span (26). A span is the number of statements between two consecutive textual references to the same identifier. Thus, for n appearances of an identifier in the source text, n-1 spans are measured. All appearances are counted except those in declare statements. If the span of a variable is greater than one hundred statements, then one new item of information must be remembered for a hundred statements until it is read again. The complexity of the program would be the number of spans at any point, i.e., the amount of data the reader must be aware of when reading any particular statement.

Control and data structure measures. There are models of structure that address the integration of control and data flow. One such model is slicing (27). Informally, slicing reduces a program to a minimal form which still produces a given behavior for a subset of the data. The desired behavior is specified as a projection from the program's original behavior. For instance, if a program computes values for variables X, Y, and Z, then one projection might be the value of X at program termination. The minimal program is obtained by eliminating program statements which do not affect the projected behavior. The result is a smaller program which contains only those statements from the original program which affect the selected behavior.

There are several possible metrics based on program slicing. These include (1) coverage, the ratio of slice length to program length; (2) overlap, a measure of the sharing of statements among different slices; (3) clustering, the percentage of statements in the slice which were adjacent in the original program; (4) parallelism, the number of almost disjoint slices; and (5) tightness, the ratio of statements found in every slice to total statements in the original program. Each of these metrics gives some view of the complexity of the program with respect to the control and data flow.

Reliability

Measuring the reliability of a product may involve an analysis of the (1) distribution or classification of errors, or (2) execution of the product in a testing or operational environment. Metrics involving the distribution of errors can include the program changes and error-day metrics discussed earlier. Other metrics involve distributions, such as fixes per line of code, fixes per phase, errors per person hour, errors per type of change causing the error, fixes per detection and correction technique, etc. Weiss (28) has studied various distributions in evaluating a development methodology by showing a profile of the error distributions made when using the methodology. Endres (29) used error classification schemes to analyze the reliability of a release of an operating system.

With regard to the operation of the program, several reliability models have been proposed in the literature (14, 30, 31, 32). Software reliability here is defined as the probability that a given software program operates for some time period without software error which is detectable by executing the code on the machine for which it was designed, given that it is used within design limits. Reliability measurement can be done for evaluation purposes as well estimation purposes. The models measure reliability as a function of calendar time, computer usage or accumulated man hours and require parameters, such as the error detection rate and the total number of errors in the system, before testing. These estimates can be based on theoretical assumptions or historical data.

A particular reliability model due to Shooman (30) is based upon a set of assumptions, such as (1) the operational software errors occur due to occasional traversing of a portion of the program in which a hidden software bug is lurking; (2) the probability that a bug is encountered in the time interval Δt, after t successful hours of operation is proportional to the probability that any randomly chosen instruction contains a bug, i.e., the fractional number of remaining bugs ϵ_r. Then the probability of a failure during time interval $(t, t + \Delta t)$, given no failures have occurred up until t is proportional to the failure rate z(t) (hazard function). Thus, the probability of failure in interval Δt, given no previous failure, is $P(t < t_f \leq t + \Delta t \mid t_f > t) = z(t) * \Delta t = K\epsilon_r(\tau)\,\Delta t$ where t_f is operating time to failure, K is an arbitrary constant, τ is the debugging time in man months, t is the operating time in hours. K can be estimated by examining the history of errors detected, e.g.,

$$K \sim \frac{\text{\# catastrophic errors detected.}}{\text{total \# errors detected}}$$

The probability of no system failure in the interval (0, t) is given by the reliability function

$$R(t) = e^{-\int_0^t z(x)\,dx}$$

assuming reliability is related to the failure rate. Assuming K and $\epsilon_r(\tau)$ are independent of operating time t we get

$$R(t) = e^{-\{K\epsilon_r(\tau)\}t} = e^{-\delta t}$$

$$= e^{-K\{E_T/I_T - \epsilon_c(\tau)\}t}$$

where ϵ_c is the number of corrected errors, E_T is the total number of initial bugs in the program and I_T is the number of instruction in the program. This implies the probability of successful operation without software bugs is an exponential function of operating time.

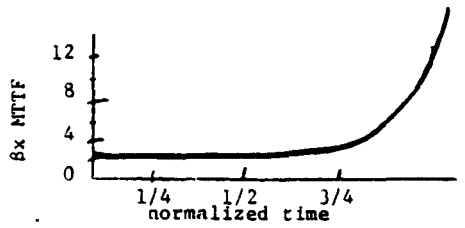
A simplier way to summarize the results of the reliability model is to compute the mean time to (software) failure, MTTF using the reliability function

$$MTTF = \int_0^{\infty} R(t)dt = \frac{1}{K\epsilon_r(\tau)} = \frac{1}{K\{E_T/I_T - \epsilon_c(\tau)\}}$$

If the error correction rate ρ_o is constant, then $\epsilon_c(\tau) = \rho_o(\tau)$ and

$$MTTF = \frac{1}{K\left[\frac{E_T}{I_T} - \rho_o t\right]} = \frac{1}{\beta(1 - \alpha\tau)}$$

where $\beta = \frac{E_T}{I_T} K$ and $\alpha = \frac{o_o I_T}{E_T}$



Note the most improvement in MTTF occurs during the last quarter of debugging.

Other models are based upon different assumptions, but all yield some measure of the reliability of the product.

The reader interested in other product measures is again referred to some general references in the literature (19, 20, 21).

PRODUCT MEASURES ACROSS TIME

As mentioned earlier, measures can be taken once on the final product or at discrete intervals throughout the life cycle. In this latter approach, metrics can be used to monitor the stability and quality of the product. By re-evaluating the metrics periodically, we can see if the product is changing its character in any way. It can provide feedback during development and maintenance. For example, if we find that over a period of time more and more control decisions have entered the system, then something may have to be done to counteract this change in character.

This approach is a way of providing a relativistic evaluation of the product. As such, it is easier to understand than an absolute measure. That is, it may be more informative to know that each change we make in the system increases the complexity of the system, than to know the total complexity of the system is some specific number. Here we need only compare the values of the metrics with values of the metrics on earlier versions of the system. The drawback to an absolute measure is that we have nothing to compare it to.

DATA COLLECTION

One major concern with performing measurement is the ability to collect reliable data. Before we begin collecting data, however, we must first understand the various factors that characterize our environment. We must isolate those factors we hope to control, measure, and understand so that we may analyze their effect.

With regard to the actual data collection process, there are various approaches. Data collection can be automated, meaning there is no interference to the developers, or non-automated, meaning the data is collected from the developers using forms or interviews. Automated data collection tends to be more reliable and can be done without the participants being aware of what specific activities and factors are being studied. Reporting forms and interviews can provide more detailed insights into the process and give a level of information that is not available in an automated collection process, e.g., insights into the kinds of errors committed.

Clearly, the data collected should be driven by the models and metrics we are interested in using; however, it doesn't hurt to add other data which may give us information about refining and modifying those models and metrics. All the data collected should be entered into a data base and validated, as much as possible, for easy reference and access.

A first step in the validation of forms is a review of the forms as they are handed in; someone connected with the data collection process should ensure that the appropriate forms have been handed in and that the appropriate fields have been filled out. The data should be entered into the data base through a program that checks the validity of the data format and rejects data out of the appropriate ranges. For example, this program can assure that all dates are legal dates and that system component names and programmer names are valid for the project by using a prestored list of component and programmer names.

Ideally, all data in the data base should be reviewed by individuals who know what the data should look like. Clearly, this is expensive and not always possible. However, several projects should be reviewed in detail and the number and types of discrepancies kept so that bounds can be calculated for the unchecked data. This allows data to be interpreted with the appropriate care.

Another type of validity check is to examine the consistency of the data base by comparing redundant data. For example, if effort data is collected both at the budget level and at the individual programmer level, there should be a reasonable correlation between the two total efforts. Another approach is to use cluster analysis to look for patterns of behavior that are indicative of errors in filling out the forms. For example, if all the change report forms filled out by a particular programmer fall into one cluster, it may imply that there is a bias in the data based upon the particular programmer.

Data collection is a serious problem, especially on large programming projects involving characteristically different environments. One set of forms may not be enough to capture what is happening across all environments. However, if we are to use this data in models and metrics, we need to know how valid that data is in each case so as to avoid improper conclusions.

CONCLUSION

Having fit the models to the data, we must analyze and interpret their results carefully. As stated earlier, we must understand the environmental parameters under which the project was developed. We must know the assumptions, strengths and weaknesses of the models in order to interpret the results for the particular project. Our level of confidence in the particular model or metric should be based upon the level to which the model or metric has been tested. If the results support our intuition, we understand what the model means in our environment; if not, understanding the model's shortcomings can yield insights into the model and our environment.

Quantitative support can be an excellent aid and risk reducer in making a difficult management or engineering decision. An organization should build up its knowledge and expertise in quantitative analysis of software development. In this way, confidence in

the various models and metrics can be acquired through direct experience.

Acknowledgement: The author would like to thank John Bailey, John Beane, David Hutchens, and Robert Reiter for their insightful review of this article. *

REFERENCES

(1) Walston, C. and Felix, C., "A Method of Programming Measurement and Estimation," IBM Systems Journal 16, Number 1, 1977.

(2) Freburger, Karl and Basili, Victor, "The Software Engineering Laboratory: Relationship Equations," University of Maryland Technical Report TR-764, May 1979.

(3) Lawrence, M. J. and Jeffery, D. R., "Inter-organizational Comparison of Programming Productivity," Department of Information Systems, University of New South Wales, March 1979.

(4) Boehm, Barry W., Draft of book on Software Engineering Economics, to be published.

(5) Doty Associates, Inc., Software Cost Estimates Study, Vol. 1, RADC TR 77-220, June 1977.

(6) Wolverton, R., "The Cost of Developing Large Scale Software," IEEE Transactions on Computers 23, No. 6, 1974.

(7) Aron, J., "Estimating Resources for Large Programming Systems," NATO Conference on Software Engineering Techniques, Mason Charter, N. Y. 1969.

(8) Carriere, W. M. and Thibodeau, R., "Development of A Logistics Software Cost Estimating Technique for Foreign Military Sales," General Research Corporation, Santa Barbara, California, June 1979.

(9) Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering 4, No. 4, 1978.

(10) Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (Ed.) Penguin Books, Inc., Baltimore, Md. 1970, pp. 77-101.

(11) Parr, Francis N., "An Alternative to the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, May 1980.

(12) Halstead, M., Elements of Software Science, Elsevier North-Holland, New York, 1977.

(13) Basili, Victor R. and Zelkowitz, Marvin V., "Analyzing Medium Scale Software Developments," Third International Conference on Software Engineering, Atlanta, Georgia, May 1978.

(14) Musa, John D., "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE1, No. 3, pp. 312-327.

(15) Belady, L. A. & Lehman, M. M., "A Model of Large Program Development," IBM Systems Journal, 1976, 15(3).

(16) Dunsmore, H. E. & Gannon, J. D., "Experimental Investigation of Programming Complexity," Proc. ACM-NBS Sixteenth Annual Technical Symposium: Systems and Software, Washington, D. C., June 1977, pp. 117-125.

(17) Basili, V. R. and Reiter, R. W. Jr., "An Investigation of Human Factors in Software Development," Computer Magazine, December 1979, pp. 21-38.

(18) Mills, H. D., "Software Development," IEEE Transactions, Syst. Eng. 2, 4 (1976), pp. 265-273.

(19) Curtis, Bill, "In Search of Software Complexity," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York, IEEE 1979.

(20) Mohanty, S. N., "Models and Measurements for Quality Assessment of Software," ACM Computing Surveys, 1979, 11, pp. 251-275.

(21) Belady, L. A., "Complexity of Programming: A Brief Summary," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York: IEEE, 1979.

(22) McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, 1976, 2, 308-320.

(23) Basili, V. R. & Reiter, R. W., Jr., "Selecting Automated Measures of Software Development," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York, IEEE 1979.

(24) Basili, V. R. & Turner, A. J., SIMPL-T, A Structured Programming Language, Paladin House Publishers, Geneva, Ill. 1976.

(25) Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974 pp. 115-139.

(26) Elshoff, "An Analysis of Some Commercial PL/1 Programs," IEEE-TSE June 1976.

(28) Weiss, David, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," Journal of Systems and Software, Vol. 1, No. 1, 1979.

(27) Weiser, Mark, "Program Slices: Theoretical, Psychological, and Practical Investigations of An Automatic Program Abstraction Method," Ph.D. Thesis, University of Michigan 1979.

(29) Endres, A., "Analysis and Causes of Errors in System Programs," Proceedings of the International Conference on Software Engineering, pp. 327-336, April 1975.

(30) Shooman, M. L. "Software Engineering: Reliability Design and Management," Note of Course EE909, Polytechnic Inst. of New York, Brooklyn, N.Y., 1976.

(31) Littlewood, B., "How to Measure Software Reliability, and How Not to . . .," in Proc. 3rd Int. Conf. Software Engineering, May 1978, pp. 37-45.

(32) Goel, A. L. & Okumoto, "A Markovian Model for Reliability and Other Performance Measures of Software Systems," Proceedings of the National Computer Conference, pp. 769-774 (1979).

*Research was supported in part by NASA Grant NSG5123

D10-61
80026

USE OF CLUSTER ANALYSIS TO EVALUATE SOFTWARE ENGINEERING METHODOLOGIES*

Eric Chen and Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

The development of quantitative measures to evaluate software development techniques is necessary if we are going to develop appropriate methodologies for software production. Data is collected by the Software Engineering Laboratory at NASA Goddard Space Flight Center on developing medium scale projects of up to ten man years effort. In this study, cluster analysis was used on this collected data and several measures are proposed. These measurements are objective, quantifiable, are the results of the methodology, and most important, seem relevant.

Introduction

Along with the development of numerous methodologies to aid in software development (e. g., structured programming, chief programmers, walkthroughs, code reading, etc.) is a growing awareness of a need to collect data to be able to quantify the effects of each new technique. Since this data is often collected after the fact, and is therefore often unobtainable and imprecise, at best, it can only be used to indicate possible trends and not specific effects of a given technique.

About five years ago, it was realized that data had to be collected as a project developed in order to better quantify a project's life cycle development. Although this imposed an additional burden on the project, it was believed that the cost was justified - both to give management more knowledge and control over the current project, and to allow the data to be analyzed later in order to determine the impact of the new techniques. Most of the recent work in this area has centered on what to collect - both in deciding what data is needed, and in overseeing the collection process to make sure that the data is collected (both manually and automatically) accurately.

This paper describes this process and a further development in this data collection trend. Now that sufficient data exists, tests are being developed to check the overall validity and value of the data. For example, if data is collected on two different projects, is there any bias in the way the two data sets are created? Can we apply the same measures on each and compare them? What techniques can be used on entire collections of data? Can we classify a project (or an attribute of the project) via measures defined on the data? These and related questions were behind the current study.

This paper introduces the concepts of cluster analysis, a well known technique in many of the social sciences, into the software development environment [Anderberg]. It shows that cluster analysis seems relevant, and the paper develops several measures that seem applicable in predicting methodologies in this environment. While the measures are based upon the data collected by the Software Engineering Laboratory, they appear to be generally applicable in a variety of settings.

At the NASA Goddard Space Flight Center in Greenbelt, Maryland, the Software Engineering Laboratory was organized in 1976 in conjunction with the University of Maryland and Computer Sciences Corporation. The purpose was to study software development within the NASA environment and develop techniques to improve software production [Basili - Zelkowitz]. Data are being collected from certain projects developed by NASA and are now under study. At present over 12,000 forms have been collected (figure 1) on some 30 projects,

* - Research supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland. Computer time provided in part by the Computer Science Center of the University of Maryland.

CH1627-9/81/0000/0117$00.75 © 1981 IEEE

Run Analysis Form	1834
Component Status Form	2669
Resource Summary	142
Change Report Form	4047
Component Summary Form	3003
General Project Summary	62
Maintenance Form	33

Figure 1. Forms collected by early 1980.

ranging in size from several thousand to over 100,000 Fortran source lines. Effort for each project varies from a few man months to about 10 man years, and most of the larger projects take about a year to complete. The programs generally provide attitude orbit information for unmanned spacecraft and operate on an IBM 360/95 computer; however, we view them simply as large application programs that include many characteristics of any software package, such as user interfaces, graphics, data base accesses, scientific computations and other characteristics.

Cluster Analysis

The information on software production is collected on a set of forms. Some forms are filled out on a regular basis. For example, the Component Status Report, filled out weekly by each programmer, gives the components of the system worked on that week, hours worked, and phase of development (e. g. design, code, test). With this data, a snapshot of the developing program can be computed, week by week. The Resource Summary gives the total hours spent by all personnel on the project for a given week.

Other forms are filled out when needed. Each computer runs results in an entry in the Computer Run Analysis form giving details of the run. Each change or correction of an error results in a Change Report Form being filled out giving the details of the change, its cause, and its effect. With this data, a complete history of a developing program can be maintained.

As each form is entered in our data base, it becomes a vector of numbers. Thus each project creates a number of data sets, where each data set can be considered a multidimensional vector space with individual forms being points in that space. Obvious questions that arise from this view are which points are near one another, does the location in the space have any meaning, and do clusters of such points have any significance?

In order to answer such questions, cluster analysis is being applied to this data. The rest of this section will describe the assumptions that we have made and the algorithms that we have used for creating clusters. The remaining sections will descibe the various applications that we have applied cluster analysis to.

In order to cluster data, the following algorithm was used:

(1) Let x and y be two points (forms) in one data set and let dxy be the "distance" between points x and y. For this study we used the similarity between two vectors via the cosine function [Salton - Wong] as our distance function since the usual Euclidean distance measures did not seem relevant to components with different characteristics. Let xi and yi be the ith selectors (data values) of vectors (forms) x and y. Then

$$dxy = \frac{\sum(xi\ yi)}{\sqrt{\sum(xi)^2\ \sum(yi)^2}}$$

dxy will have some value between 0 (if x and y are dissimilar) and 1 (if x and y are similar).

(2) Choose some threshold value B with B between 0 and 1. We assumed that for dxy<B, x is sufficiently dissimilar to y and therefore x is unrelated to y. If dxy≥B, x and y will be considered to be related. Later we will describe the various ways of choosing B.

(3) Compute the connectivity matrix C such that

$$Cxy = \begin{cases} \underline{true} \text{ if } dxy \geq B \\ \underline{false} \text{ if } dxy < B \end{cases}$$

Cxy=true means that nodes x and y are near one another and are considered to be connected. Since dxy=dyx, C is a symmetric matrix. We have now converted the distance matrix into a graph-structured connectivity matrix. Cxy=true means that there is an arc from node x to node y in some subgraph of all nodes. It is only necessary to compute the total subgraph of connected nodes in order to arrive at the cluster.

(4) The connected subgraph can be computed by computing the transitive closure C* of C:

$$C^* = C + C^2 + C^3 + \ldots + C^n$$

where addition and multiplication refer to the logical operations of "or" and "and", respectively. In this case, C*xy = true if and only if x and y are in the same connected subgraph.

The set of subgraphs forms a disjoint set of clusters. Every point belongs to one and only one (possibly singleton) cluster.

This algorithm was used to cluster based upon subsets of the possible selectors for each vector. Various other criteria (e. g., an additional selector not used in clustering) were used to see if they were predictors of which cluster a given form would reside. If so, then this selector is a dependent variable, and relating back to the original goals of the research, would indicate a relationship between the methodology (as specified in the criteria) and the data collected on the forms.

Development Histories

Current software folklore states that better systems result if a greater emphasis is placed on design. Each such report gives its own correct formula (e. g. 40% design, 20% code, 40% test), but very little quantitative data exists for verifying such relationships. Most studies basically state that "we did it this way and the results look good." As an initial test of cluster analysis we decided to investigate this question. This would also be a relatively easy test on the merits of cluster analysis itself as a valid measuring tool in our environment.

In our data base we collect the number of hours each programmer spends each week on each component. A component roughly translates into a Fortran subroutine . The stage of development worked on (design, code, test) is also indicated (figure 2). (The group that we are monitoring at NASA gets the specifications from another group and a third group takes over the software for its operational lifetime. Thus we are only evaluating the actual development process.) For this reason, the percentages that we develop later in this paper differ from more "classical" life cycle models, since we are mostly ignoring requirements, specification and operational phases.

Due to high computer costs, we limited ourselves to the 50 largest components (out of about 400) per project assuming that most of the effort on a project will be used to build the largest components. These will have a greater influence on the overall methodology than the others. The largest component needed about 400 hours to complete while the smallest of the 50 required about 25 hours.

Assuming a continuous curve, smoothing techniques were applied. In order to compare dissimilar components, the time axis was converted from weeks into deciles and the effort (vertical) axis was converted into per cent of total effort. Thus any two components were comparable (figure 3).

WEEK	DESIGN	CODE	TEST
1	33.0	0.0	0.0
2	40.0	0.0	0.0
3	42.0	0.0	0.0
4	89.0	0.0	0.0
5	1.0	0.0	0.0
6	10.0	0.0	0.0
7	7.0	0.0	0.0
8	10.0	2.0	0.0
9	0.0	2.0	0.0
10	0.0	8.0	6.0
11	4.0	6.0	1.0
12	1.0	2.0	4.0
13	0.0	6.0	4.0
14	0.0	5.0	2.0
15	0.0	0.0	0.0
16	1.0	1.5	7.0
17	0.0	0.0	6.0
18	0.0	3.0	18.0
19	0.0	0.0	40.0
TOTAL	238.0	35.5	88.0

(a)

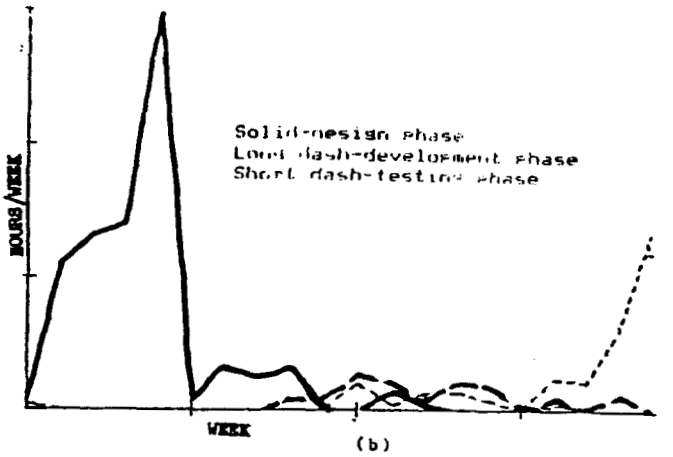


Figure 2. Effort (in hours) to develop a typical component (by week)

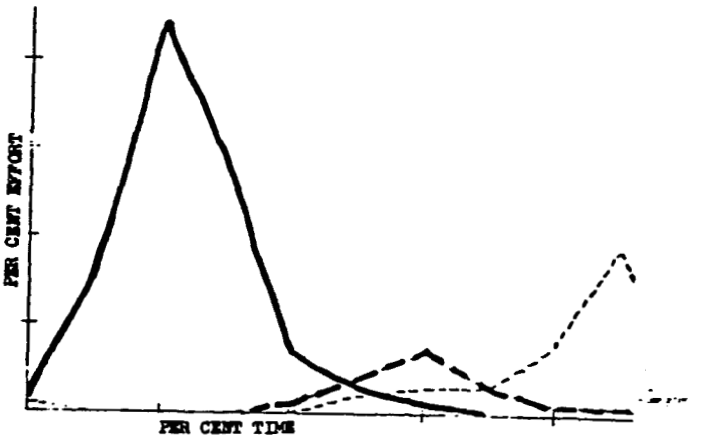


Figure 3. Smoothed and scaled data of figure 2.

Module Modifiability

In order to pick an appropriate B, various values were tested on five projects (figure 4). As B varied between 0 and 1, 4 of the five tested project had similar number of clustered components. Only project E differed and project E was the only one of the five that consisted mainly of reused modules from similar previous projects. Thus the number of clusters, relative to B, may be an invariant that can be used to measure the "newness" of the source code. Such a measure can be objectively applied to a given project to determine the degree to which previous source code has been modified for this new project.

We used a B that forced the largest cluster to be under 20 components in size. A smaller B caused many of the clusters to merge into one large cluster while any larger B caused cluster size to drop rapidly.

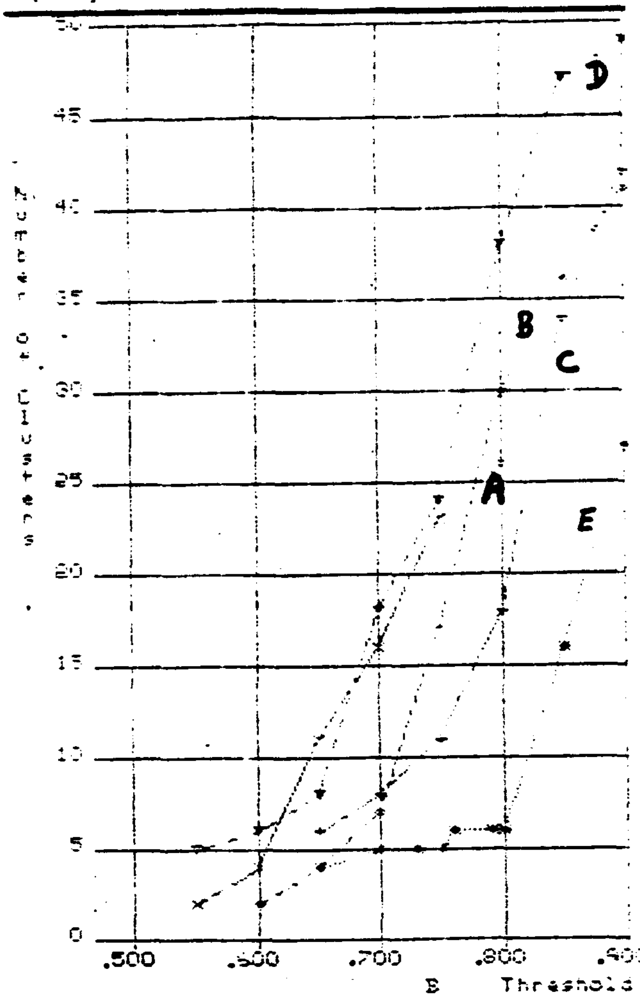


Figure 4. Number of clusters as B varies (for projects A, B, C, D, and E)

Project A

UNCHANGED	CHANGED
6	2
0	5
0	2
0	3
0	10
0	2

Unchanged: 10

Project B

UNCHANGED	CHANGED
2	5
1	5
1	2
1	1
1	1
0	2
0	2
0	4

Unchanged: 10

Project C

UNCHANGED	CHANGED
3	1
2	0
2	0
1	1
0	2
0	2

Unchanged: 32

Project D

UNCHANGED	CHANGED
3	0
2	0
1	1
0	2
0	2
0	2
0	2
0	5

Unchanged: 12

Project E

UNCHANGED	CHANGED
3	1
1	15
0	3
0	3
0	2

Unchanged: 8

Figure 5. Clusters. Each line represents one cluster of changed and unchanged components

Module Correctness

Each component's development history was now reduced to 30 values (3 at each 10 percentile), and these 30 values were used to cluster the 50 components in each project. As an independent variable we considered whether a component had been modified via a change or error. This would be a measure of how effective the process had been. Once unit testing is completed, a component is added to the project's library. If it ever changes after that date (due to further testing), then a change report form is submitted. We simply looked for change report forms that had been filled out for the 50 components under study.

Approximately 80% (about 40) of the 50 components for each project are eventually altered (figure 5). However, in 4 out of 5 projects, all of the unaltered components seem to merge into a few clusters that contain few (if any) altered components. Thus the shape of the development history curve seems to be an indicator of component reliability (as measured by the absense of any changes during testing). The physical significance of each "error free" cluster is now under study.

Projects A B D E

UNCHANGED	CHANGED
4	1
4	0
2	2
1	9
1	1
1	1
1	6
1	2
0	2
0	2
0	2
0	2
0	2
0	2
0	2
0	2
0	2
0	2
0	3
0	3
0	3
0	3

Figure 6. Clustering 4 projects

The reliability of this conclusion was enhanced somewhat by merging all four projects and clustering the 200 resulting components. In this case the error free clusters did seem to merge (figure 6).

Phase effort

We were now ready to test one of our original hypotheses - per cent of effort in each phase. Project D had to be eliminated since the data collection began when this project was mostly complete with design.

For project A, the cluster with 6 out of 8 unchanged components turned out to have: Design: 64.1%, Code: 14.4%, and Test: 21.4%, while the five clusters with errors broke down as follows:

DESIGN	CODE	TEST
4.5	53.3	42.1
0	78.7	21.2
7.3	51.7	40.8
0	34.1	65.8
0	35.4	64.5

which clearly shows the value of good design efforts.

The data for projects C and E have similar results:

PROJECT C

UNCHANGED CLUSTERS			CHANGED CLUSTERS		
DES	CODE	TEST	DES	CODE	TEST
83.6	16.3	0	37.3	40.2	22.3
50.9	45.0	3.9	21.4	50.0	28.5
81.7	3.4	14.8			
100.0*	0	0			

*- This shows that while we believe that the data is accurate, some errors must exist.

PROJECT E

UNCHANGED CLUSTERS			CHANGED CLUSTERS		
DES	CODE	TEST	DES	CODE	TEST
97.5	2.4	0	7.3	83.5	9.0+
			25.2	67.5	7.2
			1.5	82.6	15.8
			24.0	75.9	0

further strengthening this result.

For project B, where clustering was not as effective, the breakdown was as follows. For clusters with at least one unchanged component:

DESIGN	CODE	TEST
22.0	66.2	11.3
27.6	52.8	19.3
26.0	39.7	34.2
94.9	5.9	0

and for clusters with changed components:

DESIGN	CODE	TEST
78.9	15.3	5.6
6.2	66.7	25.9
22.4	24.1	53.4

and do not seem to have much significance. Project B, interesting enough, was the one project that had the hardest time meeting its objectives.

In order to put these numbers in perspective, for the NASA environment, the per cent design, code and test effort was computed. If the data is displayed in the conventional manner using official milestone dates for each phase (figure 7a), then design accounts for about one quarter of the effort, and code for about one half. However, if the actual phase effort is computed independent of the date the task is performed, then the percentages change significantly. Design increases about 10% and coding drops about 5%. Thus in the NASA environment, simply using milestone dates results in:

(1) Underestimating the actual design effort, and

(2) Overestimating the actual code effort.

One other aspect of this data can be seen by comparing the per cent of a task that was performed after its official milestone date (or before in the case of testing) (figure 7b). Note that a consistent 23%-25% of the design occurred during the coding phase and up to half of the testing occurred before the official test phase. Since module unit testing was considered to be part of the development phase (for figure 7), this seems significant. In addition, since project B was the most behind schedule, the 38% of design that occurred during coding might indicate a too early design milestone which

+- The cluster with one unchanged and 15 changed components was considered a changed cluster for this chart.

caused other problems later. Thus using milestone dates for phase determination must be viewed with caution.

Error Histories

A second test of cluster analysis was performed by analyzing the change report form, mentioned previously. Unlike the previous study on component development histories, where each data point represented 30 related attributes (per cent effort), the change report form consists of approximately 50 items that seek to identify a source program error, its cause, effects, and effort used find and correct the error. Figure 8 lists the selectors we used to cluster each form.

It was assumed that each set of responses on one form indicated a technique used to debug a system. Therefore the set of forms could be as a measure of the methodology used. It was assumed that different projects using different methodologies would have different clusters of change report forms.

In one run the programmer was the independent variable (i. e., not used in cluster analysis). This was to determine any bias in the way different programmers filled out the forms. However, to the .05 significance level, all programmers were uniformly distributed in all clusters. The conclusion therefore seems to be that in our environment, all programmers are doing essentially the same job. This would indicate that there is no real chief programmer/programmer dichotomy in the tasks we measured. This agrees with the subjective conclusions about these projects.

PROJECT	BY DATE			BY PHASE		
A	22.7	49.6	27.6	30.7	44.7	24.5
B	22.2	68.2	9.5	34.1	45.6	20.2
C	27.4	61.6	11.0	36.8	48.7	14.5
E	30.2	52.3	17.4	42.0	50.4	7.6

(a) Per cent design, code and test by milestone date and actual task

	%DESIGN DURING CODE	%CODE DURING TEST	%TEST DURING DESIGN & CODE
A	23	27	49
B	38	4	67
C	25	8	56
E	25	21	24

(b) Per cent effort during another phase

(Data collection began after the design phase of project D, so it is omitted here.)

Figure 7. Project task breakdown by date and phase

Dates (time error found, fixed)
Type of error
Time to make and fix change
Causes of error
Tools to find error
Was error related to other errors
Time to locate error
Clerical error

Figure 8. Sample data used in change report

Methodology Signatures

The set of clusters for an entire project define the basic methodology for developing a software project. We call this set of clusters its methodology signature. Two similar projects using similar techniques should have similar signatures. That is, they should find each type of error in approximately the same ratio using similar techniques for the discovery.

To test this we clustered the change report forms of several projects, then we combined the forms for two of them and clustered the merged set. Each cluster in the merged set represented two clusters - one from each set. We counted the number of components in each cluster and graphed these (figure 9). Note that large clusters tended to merge with large clusters and small clusters with small clusters. The merged set of clusters had a correlation coefficient of .32 with respect to the clusters that make up the set.

This leads to an interesting methodology measure. First cluster two of the sets of change forms and then look at the clusters formed by clustering the combined set of forms. If they have similar patterns and similar clusters merge together, then they indicate similar development structure and

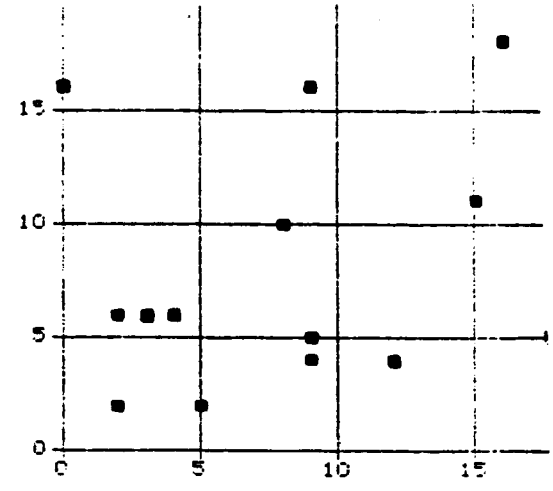


Figure 9. Number of components from project A (horizontal) and project B (vertical) in joint clusters.

probably similar methodologies. If not, then further study is indicated. Either the methodologies differ, or the class of errors found differ for some reason.

An interesting idea (although only speculation at this time) would be to generate a set of benchmark projects each representing a different methodology. An unknown project could then be clustered with each, and the one for which the merged graph generates the highest correlation would represent the unknown methodology. If this turns out to be true, then this technique would represent a quantitative method to measure a software methodology.

Conclusions

Cluster analysis has been applied on data collected by the Software Engineering Laboratory on several projects. The preliminary results should that the technique is effective in determining characteristics about the projects and the underlying methodology used in their development. Several measures seem interesting and are now under study:

(1) The threshold value in determining connectedness of the underlying graph structure (called B in this report) seems to have significance and seems to be a measure of the "reusability" of the existing source code in a new project.

(2) The development history is an indication of probable program reliability.

(3) The methodology signature developed from analyzing the change report forms looks like an effective measure of the techniques used in developing projects.

(4) More complex measures of distance between points are being considered. The current one has the virtue of being easy to program but has the disadvantage that long threadlike "snakes" of points will be in the same cluster, rather than some central "centroid" with only points near than centroid being in the cluster.

The entire software development methodology area is often filled with vague statements, folklore, and lack of quantifiable data. It is hoped that techniques such as described here can be used to give this important topic a more quantifiable, exact and scientific footing.

Acknowledgement

uld like to acknowledge the on of Mr. Warren Miller in many of the values that are erein.

References

[Anderberg] Anderberg M. A., Cluster Analysis for applications, Academic Press, New York. 1973.

[Basili - Zelkowitz] Basili V and M Zelkowitz, Resource estimation for medium scale projects, Third International Conference on Software Engineering, Atlanta Georgia, 1978.

[Salton - Wong] Salton G and A Wong, Generation and search of clustered files, ACM Transactions on Database Systems 3, No. 4, 1978, pp. 321-346.

BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-Originated Documents

Software Engineering Laboratory, SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu, D. S. Wilson, and R. Beard, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, January 1978

†SEL-78-002, FORTRAN Static Source Code Analyzer (SAP) User's Guide, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, May 1982 (preliminary)

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson, B. Chu, and G. Page, September 1978

†This document superseded by revised document.

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, S. R. Waligora, and A. L. Green, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and F. E. McGarry, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, C. E. Goorevich, and A. L. Green, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker, C. E. Goorevich, and A. L. Green, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, P. Liebertz, et al., May 1980

SEL-80-004, System Description and User's Guide for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, W. J. Decker, J. G. Garrahan, et al., October 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-001, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., September 1981

SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, F. E. McGarry, et al., September 1981

SEL-81-003, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, G. Page, et al., September 1981

†SEL-81-004, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., September 1981

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

†SEL-81-005, Standard Approach to Software Development, V. E. Church, F. E. McGarry, G. Page, et al., September 1981

SEL-81-105, Recommended Approach to Software Development, S. Eslinger, F. E. McGarry, V. E. Church, et al., May 1982

SEL-81-006, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, December 1981

†SEL-81-007, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, A. L. Green, et al., February 1981

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, W. A. Taylor, et al., February 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker, A. L. Green, and F. E. McGarry, March 1981

†This document superseded by revised document.

SEL-81-010, Performance and Evaluation of an Independent Software Verification and Integration Process, G. Page and F. E. McGarry, May 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, Software Engineering Laboratory, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-82-001, Evaluation and Application of Software Development Measures, D. N. Card, G. Page, and F. E. McGarry, July 1982

SEL-82-002, FORTRAN Static Source Code Analyzer Program (SAP) System Description, W. Taylor and W. Decker, August 1982

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo and S. Eslinger, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-Related Literature

Anderson, L., "SEL Library Software User's Guide," Computer Sciences-Technicolor Associates, Technical Memorandum, June 1980

††Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Banks, F. K., "Configuration Analysis Tool (CAT) Design," Computer Sciences Corporation, Technical Memorandum, March 1980

---

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

††Basili, V. R., "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

††Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1980

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

††Basili, V. R., and J. Beane, "Can the Parr Curve Help with Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

††Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

††Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

Basili, V. R., and T. Phillips, "Validating Metrics on Project Data," University of Maryland, Technical Memorandum, December 1981

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

††Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

††Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

††Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis to Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Church, V. E., "User's Guides for SEL PDP-11/70 Programs," Computer Sciences Corporation, Technical Memorandum, March 1980

Freburger, K., "A Model of the Software Life Cycle" (paper prepared for the University of Maryland, December 1978)

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Hislop, G., "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

Lange, S. F., "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

Miller, A. M., "A Survey of Several Reliability Models" (paper prepared for the University of Maryland, December 1978)

---

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (proceedings), March 1980

Page, G., "Software Engineering Course Evaluation," Computer Sciences Corporation, Technical Memorandum, December 1977

Parr, F., and D. Weiss, "Concepts Used in the Change Report Form," NASA, Goddard Space Flight Center, Technical Memorandum, May 1978

Perricone, B. T., "Relationships Between Computer Software and Associated Errors: Empirical Investigation" (paper prepared for the University of Maryland, December 1981)

Reiter, R. W., "The Nature, Organization, Measurement, and Management of Software Complexity" (paper prepared for the University of Maryland, December 1976)

Scheffer, P. A., and C. E. Velez, "GSFC NAVPAK Design Higher Order Languages Study: Addendum," Martin Marietta Corporation, Technical Memorandum, September 1977

Turner, C., G. Caron, and G. Brement, "NASA/SEL Data Compendium," Data and Analysis Center for Software, Special Publication, April 1981

Turner, C., and G. Caron, "A Comparison of RADC and NASA/SEL Software Development Data," Data and Analysis Center for Software, Special Publication, May 1981

Weiss, D. M., "Error and Change Analysis," Naval Research Laboratory, Technical Memorandum, December 1977

Williamson, I. M., "Resource Model Testing and Information," Naval Research Laboratory, Technical Memorandum, July 1979

††Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York:

Computer Societies Press, 1979

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982